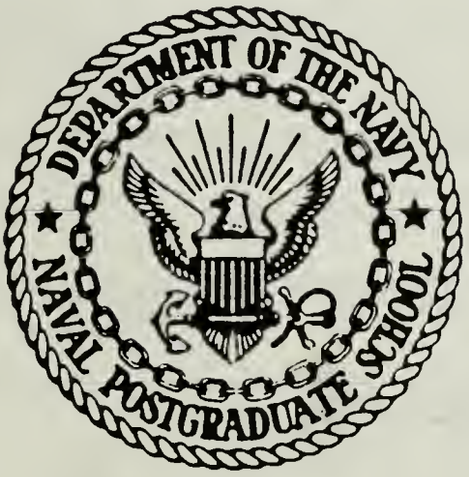
NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

SOFTWARE TOOL SELECTION FOR A
U. S. NAVY SOFTWARE MAINTENANCE ORGANIZATION

by

Joanne Sexton

June 1987

Thesis Advisor: Gordon H. Bradley

Approved for public release; distribution is unlimited

UNCLASSIFIED
ECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; Distribution is unlimited
b DECLASSIFICATION/DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
a NAME OF PERFORMING ORGANIZATION aval Postgraduate School 6b OFFICE SYMBOL (If applicable) Code 52	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
ADDRESS (City, State, and ZIP Code) onterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
NAME OF FUNDING/SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

TITLE (Include Security Classification)
OFTWARE TOOL SELECTION FOR A U.S. NAVY SOFTWARE MAINTENANCE
RGANIZATION (u)

PERSONAL AUTHOR(S)
Sexton, Joanne

TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
aster's Thesis	FROM _____ TO _____	1987 June	127

SUPPLEMENTARY NOTATION

COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Program understanding; Software maintenance; Software tools; Software environments

ABSTRACT (Continue on reverse if necessary and identify by block number)

Software tools have been in existence for a number of years. The term oftware environments, or how well software tools work together, has been a current topic in the literature. Unfortunately, those discussions have en limited to software production environments only. A greater need tists to define what is required in a software maintenance environment. oftware maintenance environment requirements should drive the needs of oduction environments because of the greater permanence of maintenance id its more sizable effect on overall software lifecycle costs. As a ep in that direction, this thesis examines one particular aspect of soft- re maintenance - how to understand programs. With this particular focus, is thesis defines criteria to rate software maintenance tool selection, id offers alternative solutions for organizational aspects that are not rrently automated by software tools.

DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
NAME OF RESPONSIBLE INDIVIDUAL of. Gordon H. Bradley	22b TELEPHONE (Include Area Code) (408) 646-2359 22c OFFICE SYMBOL Code 52Bz

ORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Approved for public release; distribution is unlimited

Software Tool Selection for a
U. S. Navy Software Maintenance Organization

by

Joanne Sexton
Lieutenant, United States Navy
B.S., Rutgers University, 1978

Submitted in partial fulfillment of
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1987

ABSTRACT

Software tools have been in existence for a number of years. "Software environments," or how well software tools work together, has been a current topic in the literature. Unfortunately, those discussions have been limited to software production environments only. A greater need exists to define what is required in a software maintenance environment. Software maintenance environment requirements should drive the needs of production environments because of the greater permanence of maintenance and its more sizable effect on overall software life-cycle costs. As a step in that direction, this thesis examines one particular aspect of software maintenance—how to understand programs. With this particular focus, this thesis defines criteria to rate software maintenance tool selection, and offers alternative solutions for organizational aspects that are not currently automated by software tools.

THESIS DISCLAIMER

The following trademarks are used throughout this thesis:

Trademark	Trademark Owner
ALL-IN-1	Digital Equipment Corporation
DATATRIEVE	Digital Equipment Corporation
DEC	Digital Equipment Corporation
DECalc	Digital Equipment Corporation
DEC/CMS	Digital Equipment Corporation
DEC/MMS	Digital Equipment Corporation
DECnet	Digital Equipment Corporation
DEC/Test Manager	Digital Equipment Corporation
DECwriter	Digital Equipment Corporation
DIBOL	Digital Equipment Corporation
MicroVAX	Digital Equipment Corporation
Micro-VMS	Digital Equipment Corporation
VAX	Digital Equipment Corporation
VAXELN	Digital Equipment Corporation
VAXcluster	Digital Equipment Corporation
VAXnotes	Digital Equipment Corporation
VAXset	Digital Equipment Corporation
VAXstation	Digital Equipment Corporation
VMS	Digital Equipment Corporation
VT	Digital Equipment Corporation

TABLE OF CONTENTS

I. BACKGROUND

A. INTRODUCTION TO SOFTWARE TOOLS AND ENVIRONMENTS

Software tools have been in use for a long time. Anyone who has used a computer to produce written documents or to code a small program has used a software tool. Word processors and programming language compilers and interpreters are examples of software tools.

Literally hundreds of software tools exist to help the software manager, designer, and maintainer to do their job better [Ref. 1:p. 21]. So many software tools exist in fact that a number of articles and pamphlets have been written just to help classify them [Refs. 2, 3, 4, and 5].

With this wide proliferation of software tools, no one can possibly know how to use all of them or even know all the tools that may exist. As a way to deal with this complexity, the concept of a programming environment or a software engineering environment has evolved. An environment is a means to collect and integrate a set of software tools into a useful whole. Charette [Ref. 1:p. 38] extends this definition of a software engineering environment to include all "the processes, methods, and automation required to produce a software system."

B. OBJECTIVES OF RESEARCH

All software organizations are interested in improving programmer productivity. The Naval Security Group Detachment Pensacola, FL Software Support Activity is no exception. (In the rest of this thesis,

this activity will be referred to as the Software Support Activity.) The Software Support Activity is a new Navy activity that has been established in Pensacola, FL to perform software maintenance. One of the prime concerns of this new organization is how to improve programmer productivity through the use of software tools.

The issues of productivity and software tools in general are too broad to handle adequately in any thesis. As a consequence, the scope of this thesis has been narrowed to look at one aspect of software maintenance—understanding software programs. The decision to look at this particular aspect is based on a study done by Fjeldstand and Hamlen [Ref. 6] that analyzed how maintenance programmers spend their time. The Fjeldstand and Hamlen study [Ref. 6] is covered in greater detail in chapter five of this thesis; one of its findings is that maintenance programmers spend over 60% of their time reading and analyzing programs. The premise of this thesis is that any means that can be found to improve programmer effectiveness in understanding programs would have a significant productivity savings for the organization as a whole. The focus of this thesis is program understanding and how software tools and environments may help this process.

In order to look at these aspects, a full understanding of the Software Support Activity, a review of the issues of software tools, an examination of software maintenance in general, an evaluation of the Software Support Activity's existing tool set, and an in-depth analysis of program comprehension must be achieved. Each of these subjects are covered in subsequent chapters. Recommendations and solutions

to help improve program comprehension through the use of software tools and environments are also presented in the final two chapters.

C. LIMITATIONS AND ASSUMPTIONS

Limited guidance exists as to what an organization should buy for a software environment. No guidance exists when the organization being considered is a software maintenance activity.

To better understand the critical issues involved in developing requirements for a software maintenance environment a specific organization (the Software Support Activity) was chosen for study. It is the hope that findings established for one organization will prove universal and will be transferable to other software maintenance organizations.

Before any discussion of software tools can be attempted, the new organization, its role and functions, must be examined. Appendix B contains a questionnaire that was developed to gain more knowledge about the Software Support Activity. The next chapter describes the new organization from information gained from the questionnaire.

II. OVERVIEW OF THE SOFTWARE SUPPORT ACTIVITY

A. INTRODUCTION

The contents of this chapter are as follows: First, we explain what the Software Support Activity is, and what its functions and responsibilities are. Second, an overview of the role of headquarters elements is given. Third, a description of the hardware and software is provided. Fourth, communication facilities are briefly detailed. Fifth, a coverage of the backgrounds of the personnel is described. Next, Software Support Activity personnel training is outlined. Finally, a synopsis of the proposed system and prospective problem areas is covered.

B. FUNCTIONS OF THE SOFTWARE SUPPORT ACTIVITY

The Software Support Activity is a Naval Security Group detachment that was officially established 6 March 1987 to perform centralized software support for shore-based cryptologic systems and related functions. It is located onboard Corry Station, Pensacola, Florida and it will be a tenant command of the Naval Technical Training Center (NTTC).

The Software Support Activity will assume software support responsibilities for SIGINT Classification of Recognition of Classified Emitters (SCORE) and the Mobile System Technical Data Facility (MSTDF) on 1 October 1988. Between November 1987 and 1 October 1988, the Software Support Activity will be learning the SCORE and

MSTDF application software and related hardware and installed commercial software packages.

SCORE is a HULTEC database system that produces reports which are consumed directly by fleet units. It is being developed by NOSC (Naval Ocean Systems Command) San Diego, California. MSTDF is a master database facility that will be used to support deployed units and is being developed by Engineering Research Associates (ERA) of McLean, Virginia. Each of these systems will be installed at various Naval Security Group operational sites worldwide.

The Software Support Activity will be performing software maintenance on the SCORE and MSTDF application software and will also make updates to any installed commercial software packages. The maintenance that the Software Support Activity will perform includes the following:

- Fixing bugs
- Making Class II or minor enhancements (Class II enhancements are any upgrade that does not concern technical, monetary, performance, specification, or schedule changes that affect configuration identification (CI) items) [Ref. 7:p. 2.22].
- Improving software performance and source code efficiency
- Any change or improvement that does not change the form, fit, or function of the system, i.e., does not need Configuration Control Board (CCB) approval
- Auditing each maintenance phase to ensure all acceptance criteria have been meet
- Testing

- Configuration management
- ADP security

The Software Support Activity organization is depicted in Figure 1. The Activity is divided into three departments: the Support Department, the Software Maintenance Department, and the Quality Assurance Department.

The Support Department's main focus is supporting the Software Support Activity. It includes an Administration Division that performs all necessary clerical functions and oversees the Software Support Activity's budget. The Information Systems Division is responsible for all installed commercial software packages and hardware. As such, they are the activity's resident DEC VAX/VMS experts. The Information System Division has nothing to do with the SCORE and MSTDF application systems but is responsible for setting up and maintaining all application libraries and procedures for using all software tools.

The Software Maintenance Department's main focus is the operational sites in the field. This department is directly responsible for the performance and efficiency of the source code and data bases used in the SCORE and MSTDF applications. The Software Maintenance Department will go to the field to resolve problems if necessary and is responsible for upgrading the field systems.

The Quality Assurance Department's focus is also on the field. This department's responsibility is to ensure no software is released without adequate testing. It performs an independent verification and validation function. Besides worrying about new releases, the Quality

Assurance Department performs configuration management, software library maintenance, problem report tracking, and auditing.

Specific functions and responsibilities of each department are further detailed in Tables 1 through 3. Tables 4 and 5 outline specific Software Support Activity responsibilities to the operational sites and the operational sites' responsibilities to the Software Support Activity, respectively.

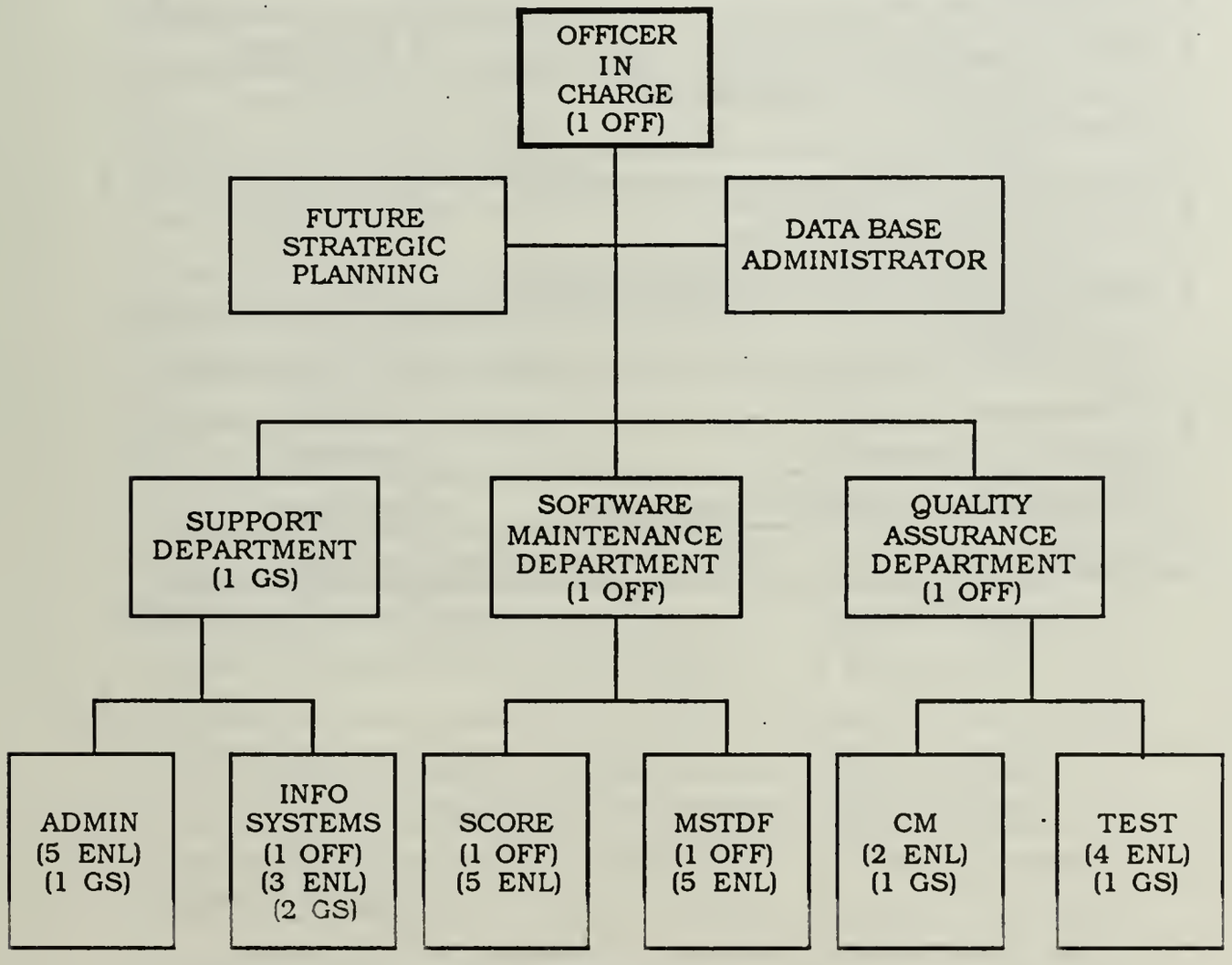


Figure 1

Software Support Activity

TABLE 1

SUPPORT DEPARTMENT RESPONSIBILITIES

Function	Division
Update & Maintain Commercial Packages	Info. Sys.
Maintain Back-up Systems	Info. Sys.
Set-up & Maintain Application Procedures	Info. Sys.
Maintain Procedures For Using Tools	Info. Sys.
Maintain Data Dictionaries	Info. Sys.
Analyze Impact of Future DEC Upgrades	Info. Sys.
Administration	Admin.
SSO	Admin.
Support Agreements	Admin.
Hardware Maintenance Contracts	Admin.
Budget	Admin.

TABLE 2

SOFTWARE MAINTENANCE DEPARTMENT RESPONSIBILITIES

Function	Division
Develop Class II Upgrades	Maintenance
Perform Software Maintenance	Maintenance
Resolve Problem Reports	Maintenance
Evaluate Change Requests	Maintenance
Conduct Training	User Liaison & Maintenance
Meet QA Standards	Maintenance
Communicate With User	User Liaison
Perform Trend Analysis & Future Planning	Trend Analysis
Field Tiger Teams	Maintenance
Maintain System Maintenance Journal	Maintenance
Maintain Error History	Maintenance
Deliver Scheduled Updates to Field Stations	Maintenance
Generate Periodic Status Report	User Liaison
Update Problem Reporting Procedure	User Liaison
Maintain Maintenance Statistics & Software Metrics	Trend Analysis
Incorporate QA Department Approved Software Changes Into Baseline	Maintenance

TABLE 3

QUALITY ASSURANCE DEPARTMENT RESPONSIBILITIES

Function	Division
Perform Configuration Management	CM
Perform Quality Assurance	CM
Train QA & Test Personnel	CM & Test
Conduct Surprise Audits	CM
Conduct Field Testing	CM
Conduct Assist Visits	CM
Verify and Validate Software Product	Test
Maintain Program History	Test
Update & Develop Test Plans/ Procedures & Data	Test
Conduct Acceptance Testing of Class II Updates	Test
Conduct Acceptance Testing of Problem Fixes	Test

TABLE 4

SOFTWARE SUPPORT ACTIVITY RESPONSIBILITIES TO OPERATIONAL SITES

Function	Site Elements Involved
Conduct Surprise Audit	On-site Software Personnel & Operations
Conduct Assist Visit	On-site Software Personnel & Operations
Install Major Software Upgrade	On-site Software Personnel & Operations
Conduct User Training	Operations
Install Minor Software Upgrade	On-site Software Personnel
Conduct On-site Software Personnel Training	On-site Software Personnel

TABLE 5

OPERATIONAL SITE'S RESPONSIBILITIES TO THE SOFTWARE SUPPORT ACTIVITY

Function	Who Performs	For Whom
Install Minor Software Upgrade	On-site SW Personnel	SSA & Operations
Perform Emergency Fix	On-site SW Personnel	SSA & Operations
Diagnose Unfixable Problems	On-site SW Personnel	SSA
Perform Configuration Management	On-site SW Personnel	SSA & Operations
Report Statistics	Operations	On-site SW Personnel
	On-site SW Personnel	SSA
Conduct User Training	On-site SW Personnel	Operations
Analyze Performance	On-site SW Personnel	SSA
Revise/Report Local Data Model	Operations	On-site SW Personnel
	On-site SW Personnel	SSA

C. HEADQUARTER ELEMENTS

Commander. Naval Security Group (COMNAVSECGRU) is the headquarters element for both the Software Support Activity and the operational sites supported by the SCORE and MSTDF systems. COMNAVSECGRU is thus classified as the user of SCORE and MSTDF

and is responsible for post development maintenance of SCORE and MSTDF and the software lifecycle support of these two systems.

SCORE and MSTDF have been developed under the control and guidance of the Space and Naval Warfare Systems Command (SPAWAR). SPAWAR is designated as the Project Management Office and is responsible to ensure that the contractors develop SCORE and MSTDF in accordance with its standards. SPAWAR fully defines contractor responsibilities and system deliverables in the Shore Cryptologic Support System Computer Resources Lifecycle Management Plan [Ref. 8].

As a further note, SPAWAR chairs the Configuration Control Board (CCB) for SCORE and MSTDF. COMNAVSECGRU, each of the contractors, and the Software Support Activity are all members of the CCB.

Figure 2 indicates the relative relationships between all of these organizations.

D. HARDWARE AND SOFTWARE

The Software Support Activity hardware consists of two VAX 8200 Programmer Workbenches, eight MicroVAX II computers, 50 VT 241 color terminals, and a VAX system to emulate the SCORE and MSTDF systems.

One of the VAX 8200s will have the following productivity tools and the FORTRAN and the Pascal compilers installed on it:

- VAX Language-Sensitive Editor
- VAX Performance and Coverage Analyzer

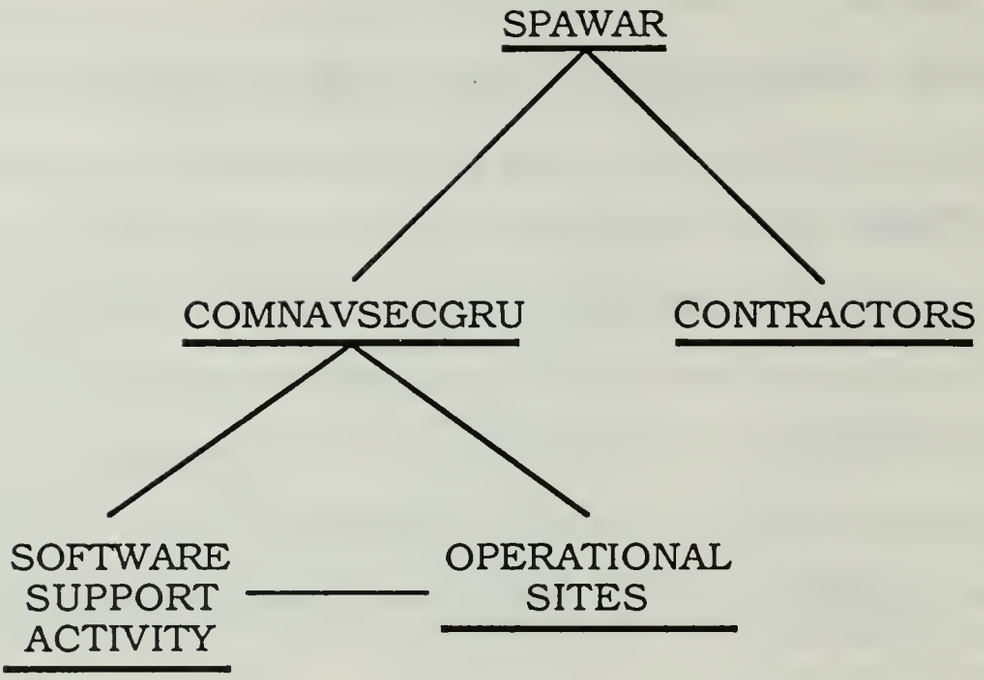


Figure 2

High-Level Organization Hierarchy

- VAX DEC/Test Manager
- VAX DEC/Code Management System (CMS)
- VAX DEC/Module Management System (MMS)
- VAX Common Data Dictionary (CDD)

The second VAX 8200 will be used as the Development System. It will host the eight MicroVAXes. Each of the MicroVAXes will be configured with both language compilers. The idea will be to use the MicroVAXes to do any necessary compilation and to uplink to the VAX 8200 to access the main libraries. The VT 241 color terminals will be installed at each desk within the Software Support Activity complex.

The Administration Division of the Support Department will have one to two ALL-IN-1 Office Automation Systems. These systems will use the WPS word processor and will include DECGRAPH.

Other Software Support Activity software will include the following:

- EDT Text Editor
- VAX Symbolic Debugger Utility
- VMS 4.4 Operating System
- Micro-VMS 4.4 Operating System
- VAX FORTRAN
- VAX Pascal
- VAX Forms Management System (FMS)
- DEC CALC
- DEC GRAPH
- DEC SLIDE
- VAX DATATRIEVE
- DECnet

Some of the software tools listed above are covered in greater detail in Appendix C.

E. COMMUNICATIONS

All systems within the Software Support Activity will be connected via a local area network (DECnet). Personnel will be able to access any system, terminal or processor, within the network through the VT 241 terminal on their desk.

Initially, data communications connectivity between the Software Support Activity and the operational sites will not be possible. It is planned for the future when one of the VAX systems will act as a host computer for a connection to the PLATFORM network (DOD computer resources network). This interface will provide worldwide access and the ability to transfer data files.

Although desirable, SCORE and MSTDF have not been built with any real thought to distributing the processing or the data bases between the operational sites. This was largely not done because each of the developers considered it too hard to accomplish.

F. PERSONNEL

The Software Support Activity will be manned by 6 officers, 24 enlisted, and 6 government service personnel. The exact distribution of each category is pictured on Figure 1.

Each of the team leaders of SCORE and MSTDF will be officers who are Naval Postgraduate School computer science graduates. Each team will be comprised of one E-7, two E-6s, and two E-5s. The hope is to augment these teams with development contractor personnel. Civilians working in the Quality Assurance Department are required to have previous Naval Security Group operational experience. Civilians hired for the Information Systems Division must have significant previous VAX/VMS experience.

Two types of personnel exist at each of the operational sites: operators and on-site software personnel. The operators have no previous software experience, but have operated highly technical

computer systems in the past. The on-site software personnel do have some software experience. They are operational site assets but will serve as an interface between the Software Support Activity and the local command.

The on-site software personnel will help diagnose problems, perform small software updates, and are authorized to make emergency software changes under strict rules and procedures.

G. TRAINING

Each of the enlisted personnel and most of the officers will have had heavy field experience. They have either worked in the exact same job as the operators who will be using SCORE and MSTDF or they have worked in a closely related job. The software development experience on the whole for all military personnel is quite limited. Extensive training to include the following is required:

- 12 week Navy FORTRAN Programming Course
- 13 week Navy System Programmer Course (teaches VAX/VMS/ ORACLE/Data Structures)
- Navy SCORE and MSTDF Operator Training Course (to support the systems, the programmers must be familiar with their operation)
- On-the-job training with the software developers (the programmers must become familiar with the software code that the contractors have developed)
- DEC on-site courses, to include System Management, Cluster Management, Device Drivers, Programming in the VAX Pascal Environment, and Performance Analysis
- A need also exists for a course in Data Resource Management—how to use data dictionaries and data directories

The Software Support Activity is not responsible for user training. It is only responsible for the software. Each operational site will ultimately be responsible for its own training. The Software Support Activity will assist and will play a role in training the on-site software personnel, however.

H. POTENTIAL PROBLEM AREAS

The best aspects concerning the development of SCORE and MSTDF are the standards and deliverables that were asked for in the Shore Cryptologic Support System Computer Resources Lifecycle Management Plan [Ref. 8].

It was specifically stated that the following procedures and standards would be used: top-down design, top-down analysis, structured appoach, emphasis on the modularity of components, top-down implementation, top-down testing, use of a Data Element Dictionary (DED) as the primary data- base design tool, and designing the data base to third normal form. [Ref. 8:pp. 11 -13, 17]

The following are considered deliverables of the system:

- Program Performance Specifications
- Program Design Specifications
- Database Design Document
- HIPO charts
- Interface Design Specifications
- Software Development Specifications (outlines the contractor's understanding of what software needs to be developed)
- Configuration Management Plan

- Quality Assurance Plan
- Computer Program Test Specifications
- Computer Program Test Procedures
- Computer Program Test Plan
- Software Development Plan (explains each contractor's approach in designing the system software) [Ref. 8:pp. 10, 12, 14, 26, 44]

In addition, the following concepts and notions were requested:

- Functional Configuration Audit
- Range of testing to include: module testing, subprogram testing, computer program performance testing, integration, and system testing
- Functional Qualification Review
- Physical Configuration Audit
- Quality Assurance Mechanism (to provide for the detection, reporting, analysis, and correction of program deficiences)
- The requirement that each module will have a well defined function with all inputs and outputs specifically identified and documented
- Source code will be checked to ensure that it is thoroughly documented with purpose comments that explain the function of each module
- Source code will be audited to ensure it meets specifications, is traceable to requirements, and conforms to coding standards and conventions
- The delivered software will be verified that it can be compiled, assembled, linked, loaded, and executed correctly from documented procedures
- The baselined documents will be evaluated for comformity, clarity, completeness, maintainability, and to ensure that they accurately represent the current software [Ref. 8:pp. 17 - 19, 22 - 26, 42]

The standards and deliverables requested are all good. They represent what should be asked for. The problem is that there is no way to enforce that the standards have been meet. The Shore Cryptologic Support System Computer Resources Lifecycle Management Plan [Ref. 8] is a guideline for the contractors to follow. There is no guarantee that the source code really has been designed with software maintenance in mind. COMNAVSECGRU has only been involved to ensure operational needs are met. Software issues to date have not been a COMNAVSECGRU concern.

Additional problems exist. Contractors were not given any specific standards to make the source code more maintainable. No test equipment (performance monitors, simulators) or test support software (scenario generators, test drivers, stub generators, test data) were requested to be developed or turned over as deliverables [Ref. 8:p. 46]. ERA is developing MSTDF with a Master Data Element Dictionary. NOSC is not developing SCORE with one. The use of Pascal could prove troublesome. Pascal is not recognized as a standard programming language for Navy use. Although the notion of quality assurance and configuration management are known, what exactly will be developed in these areas is an unknown. The Software Support Activity expects to write its own Quality Assurance Plan and Configuration Management Plan. No Software Support Activity representation has been involved in any of the reviews or plans for SCORE and MSTDF. Although SCORE and MSTDF are both database systems, they were developed without the use of data models. The requirements of

automated systems must change to ensure that data models are required and a deliverable. For a non-computer user, SQL (The name of the relational database language used by SCORE and MSTDF) is difficult. Although most users can write queries, new users do have problems writing efficient queries. The potential exists for SCORE and MSTDF performance to be diminished until users become proficient enough to write optimized queries. Ths Software Support Activity will have to redo SCORE and MSTDF to allow availability of system-wide data dictionaries and directories, strategic data planning, data models (as previously mentioned), subject data bases, and auditing. The lack of any planning by the contractors to allow future distributed connectivity is a serious short fall. Users are going to want to back up their systems and query remote data bases. These capabilities will be extremely hard, if not impossible, to add on to the base systems at a later date.

III. CORE ISSUES OF SOFTWARE TOOLS

A. INTRODUCTION

The previous chapter gave an overview of the Software Support Activity. Although all of the activities are important, the focus of this thesis, as previously described in the introduction chapter, is to examine software tools. When one speaks of using software tools, two core issues emerge. The first is the notion of an environment, that is, how well the tools selected work together. The second is defining what the user really needs. Each issue is covered in subsequent sections of this chapter.

B. ENVIRONMENT

This section begins with a general discussion of the problems of software tools. Next it outlines the general requirements needed in a software environment. Finally, it rates, using the criteria cited in sub-section two, the Software Support Activity's tool set as an environment.

1. The Problems of Software Tools

The major problem with software tools is that they quite often are not integrated to form a useful whole. A number of extremely powerful software tools exist and have been in use for a number of years. The problem is one tool cannot be used in concert with another tool. Software tools in general are incompatible and lack uniformity. They are often language dependent and machine

dependent. Even when software tools are installed on the same computer system and the tools are based on the same programming language, they still cannot be used together. [Ref. 9: p. 405]

What is needed is the creation of a software environment. The ultimate goal of a software environment is to allow software tools to be fully integrated.

Software tools are integrated if they share a standard representation so they can communicate. To do so, they must share common data structures and a common data base. In essence, what occurs is that the output from one utility, or tool, is the input to another facility without translation.

Another desirable quality in an environment, although not absolutely necessary, is the concept of non-modal. Environments are either classified as modal or non-modal. Modal environments are more conventional. They allow you to be in one, and only one, mode at a time. If you need to use another tool, you must get out of the current mode you are in and enter a new mode. Non-modal environments allow you to remain within the context of one software tool while using the facilities of another. For example, if you are using a debugging tool to debug a program and want to make some editing changes, you can use the facilities of the editor without ever leaving the debugger. To the user, there is no difference in capabilities or view when he switches from the debugger to using editing commands. There is no action or conscious change needed to shift from one tool to another

and back again. To the user, it's just as if all tools are available within the same context or framework.

2. General Requirements for a Software Environment

Buxton and Druffel in their "Rationale for STONEMAN" [Ref. 10] give a brief synopsis of what general requirements are needed in a good software environment. They are as follows:

- Provide a well-coordinated set of useful tools. The tools must be fully integrated into a consistent environment. Tools must be able to communicate with each other. Use of a subset of the tool set, selected to match a particular user's working style, is desirable.
- Provide a consistent programmer interface. Interfaces should be consistent and similar. Related functions across different tools should be expressed in similar terminology.
- Be easy to use, easy to understand, and have a helpful user interface.
- The software environment must easily adjust to and recover from user and system errors. Meaningful diagnostic information also should be provided to its users.
- Assist various levels of programmer ability.
- Easily allow the addition of new tools and the improvement, update, or replacement of existing tools.
- Must enhance software quality issues of reliability, performance, evolution, maintenance, and responsiveness to changing environments.
- Provide a consistent environment from machine to machine, from project to project.
- Support the entire software life cycle. Software tools developed must meet the needs of the developer, maintainer, and manager.

3. Rating the Software Support Activity Tool Set as an Environment

Using Buxton's and Druffel's requirements [Ref. 10] presented in the previous section, both good and bad aspects can be identified in rating the Software Support Activity tool set in an environment (Note: See Appendix C for more detailed information on some of the tools that comprise the Software Support Activity tool set).

The VAX Software Engineering Tools' (VAXset) greatest attribute, especially when compared with other software tools commercially available, is its level of integration. VAXset tools were designed and built to work together. They are completely compatible and, as a result of being designed to a common specification, the tools can freely communicate with each other. They comprise a non-modal environment. While working in the context of one tool, the facilities of other tools may be called and used without ever leaving the scope of the original calling tool.

VAXset tools not only work and communicate with each other but they also provide a consistent programmer interface. All the tools share a common user interface and provide a consistent response to the user. As a result, the command language, prompts, and error messages used are the same across all the tools.

VAXset tools for the most part are easy to use and easy to understand. This is largely true because of the VAXset's high level of integration and sharing of a single, common user interface. The user

only has to learn one way to interface with the computer rather than learn a different interface for each tool he uses.

The VAX Language-Sensitive Editor provides extensive on-line help facilities and programming assistance by providing pre-formatted templates to help program development. The VAXset thus assists various levels of programmer ability because the programmer determines how much on-line help he requires and to what degree he wants to use the language-sensitive templates. Regardless of ability level, the Language-Sensitive Editor helps the programmer generate correct source code the first time.

The other VAXset tools help the programmer automate hard to understand and difficult tasks. Version control and the tracking of changes are made easier with the use of VAX DEC/CMS. VAX DEC/MMS helps make the building of systems easier; however, in VAX DEC/MMS, the setting of description files is awkward and complicated. Every time a system needs to be built or rebuilt, or a programmer needs to use a specific version of the system, a description file must be created. A description file defines what programs and files must be linked and loaded to define the "system" being currently worked on. This must be redone or recreated each time VAX/MMS is used. Description file creation should be automated so that it remembers what has been done in the past and is editable to allow minor updates and changes.

The VAX Performance and Coverage Analyzer enables the programmer to make performance and test coverage analysis routine

parts of everyday program development efforts, rather than a separate task completed after the code has been totally developed. The DEC/Test Manager provides an enormous assist to the programmer by standardizing routine tests the programmer should run to see if his code is consistent with already existing software and if it matches organization standards. The power of this standardization is that the test experts within an organization can design the required tests leaving the programmer more time to focus on writing code.

The VAX Symbolic Debugger is well respected within the software industry. It is totally integrated with and is used in context of the Language-Sensitive Editor. The Symbolic Debugger provides on-line help and its diagnostic information is easy to understand.

VAXset easily allows the addition of new tools. The Source Code Analyzer, a new software tool, has recently been added. The Source Code Analyzer is totally integrated into the previous version of the VAXset and it adds a significant dimension when it is used with the Language-Sensitive Editor and the Symbolic Debugger. (Note: see Appendix D for more details on the Source Code Analyzer.)

Users do not need to buy the entire tool set. Tools can be bought and used independently or added as desired. In addition, existing tools in the VAXset can be customized and extended to meet user requirements. For example, the Language-Sensitive Editor templates can be customized to match an organization's programming standards.

The VAXset was specifically designed to increase programmer productivity, increase product quality, help manage complexity, and increase the effectiveness with which programmers implement, test, and manage programs. The VAX Performance and Coverage Analyzer specifically addresses the performance issue. It enables the fine tuning and optimization of source code for peak efficiency. The VAX Performance and Coverage Analyzer will identify performance hot spots, locations in code which because of heavy use are likely candidates for improved performance. The VAX Performance and Coverage Analyzer and DEC/Test Manager address the reliability, evolutionary, and maintenance aspects of code. These tools together ensure new code remains within performance standards of already existing code. DEC/CMS helps users respond to changing environments. DEC/CMS will track all changes to code and enable the reconstruction of prior versions of the software system.

VAXset is not portable from machine to machine. It was never built to be. The tools of the VAXset are not the limiting factor in this regard. The fact that VAXset was built around the VAX/VMS operating system and VAX architecture is the restriction.

Although VAXset is not portable between computers, it is portable among sixteen different programming languages and across different kinds of projects. Since VAXset was not built around a single language, there is no need to maintain several incompatible support environments for each application language used. An added advantage of portability between different languages is that programs written in

one VAX supported language can call programs written in another VAX language. How useful this multi-language capability will be is still under a great deal of debate along with the whole related issue of software reusability. It is believed that this capability may prove useful in the transition to ADA. [Ref. 11]

VAXset was not designed for any particular kind of project. The tools are generic enough that they fit the needs of most projects.

As previously stated, VAXset's greatest attribute is its level of tool integration. VAXset's support of the entire software life cycle, on the other hand, is its weakest aspect. This is not to say that VAXset does not support life-cycle issues. It does. For example, the DEC/Test Manager automated regression testing can be used, and should be used, throughout the software life cycle. It will ensure new code written is adequately tested and fits within the existing software systems. The problem is that VAXset provides no tools that automate the front-end of the software life cycle. VAXset has no software tools that support analysis and design. It provides no means to tie software changes to a project's original specification. As a consequence, although a degree of configuration management exists in VAXset, it is less than desired. A tie between all phases of the software life cycle is needed. What VAXset does is emphasize the automation of programmer related tasks– those tasks that deal with the implementation of source code.

In terms of Buxton's and Druffles's requirements [Ref. 10], VAXset overall is an outstanding environment. Considering Digital's

support services, the fact that DEC has produced industrial strength tools, and the degree of tool integration achieved make VAXset a better environment than most UNIX- and LISP-based environments for an organization involved in software production and maintenance.

C. DEFINE WHAT THE USER NEEDS

Determining what a user needs is critical in improving a process or an organizational system. This is regardless of whether computer automation is even being considered as part of the solution. What the user needs is a function of the user's experience and ability level, the tasks the user must perform, and the conditions under which the user must work.

At the present time, no clear idea exists of what the user needs in a software environment. All current work on software environments has centered around the issue of tool compatibility. Taking a top-down approach and pre-defining the specific tools a user needs in an environment is typically not done.

Coupled with this lack of a top-down approach to environment design, the environments that have been developed are software development environments. To date, software maintenance environments have not been developed or defined. Only environments emphasizing the development portion of the software life cycle have been created to any useful degree. The need for state-of-the-art technical tools, however, is just as important to software maintenance as it is to software development activities. In fact, it is more important

because maintaining software is a more difficult task than developing the original software. [Ref. 9:pp. 404 - 405; Ref. 12:p. 138]

The subsequent chapters will help lay the groundwork to develop some requirements for a software maintenance environment.

IV. PERFORMING SOFTWARE MAINTENANCE

A. INTRODUCTION

Before the problems of a software maintenance organization can be fully explored, an understanding of software maintenance in general must be achieved. For purposes of this thesis, only source code maintenance will be examined in any detail. The additional functions such as version control, quality assurance, etc. described in chapter II are impacted by the basic process of program maintenance. These functions will be described further in this thesis but only in regard to the corollary role they play in program maintenance. With that in mind, this chapter will provide an in-depth look at software maintenance as it applies to performing the single function of source code maintenance. As such, this chapter is divided into two parts. First, a definition of software maintenance will be provided to give the reader a feel for the type of source code changes made in performing software maintenance. Second, the eight steps of performing program maintenance will be outlined.

B. DEFINITION OF SOFTWARE MAINTENANCE

Software maintenance activities are divided into three basic categories:

- Corrective Maintenance
- Adaptive Maintenance
- Perfective Maintenance [Ref. 13:pp. 492-497].

Corrective maintenance deals with the identification and correction of software errors and performance deficiencies. Adaptive maintenance involves changes needed to allow the software system to adjust to changes in the outside operational environment. Perfective maintenance is not limited to just minor changes. It is maintenance performed to make the system better, to enhance its capability and performance, and to improve the documentation and software. It is performed to enhance performance, improve cost-effectiveness, improve processing efficiency, or improve maintainability. [Ref. 9:p. 22]

C. EIGHT STEPS OF PERFORMING PROGRAM MAINTENANCE

The eight steps of performing program maintenance are as follows:

- Understand the problem
- Understand the documentation
- Understand the source code
- Modify the code
- Debug
- Test
- Perform regression testing
- Update documentation

Regardless of which of the three basic categories of software maintenance is being performed (corrective, adaptive, or perfective),

each of the eight steps applies to some degree. Each of the steps will now be fully described as subsequent sub-sections of this section.

1. Understanding the Problem

Understanding the problem is not limited to just software development. A software maintainer, like a software developer, must understand all needed requirements and functions of a new capability. In addition, a software maintainer must be able to conceptualize a problem a user is experiencing in the operation of a system and understand it in terms of the user's language and understanding. [Ref. 14:p. 115].

Understanding what a user wants is an extremely difficult task. Each software product and task must be understood by many people. Each of these people has a unique viewpoint, degree of software sophistication, and interests. A common language for communication does not exist for the varied backgrounds and experiences encompassing the large number of people involved in the software maintenance process.

Understanding the problem, or user needs, is easier in software development than in software maintenance. The developer must determine what a user wants. Based on his interpretation, he develops a software product that a user reviews to determine if the developer has understood the user's needs. If the developer's interpretation is accurate, then the developer proceeds with analysis, design, implementation, and test phases of the software life cycle. If

not, then the developer re-works his interpretation of the problem and resubmits the new version for further user review.

In contrast, the maintainer requires a more exact definition of the problem. If the user has reported an operational bug, then the maintainer must be able to duplicate the precise error. He must, also, understand the error in terms of its execution complexity and its relation to the rest of the software system.

When a maintenance programmer is designing a software change, he follows the same software life cycle development steps as the software developer. The maintainer's understanding of the problem must make his software change fit within an existing software system. The maintainer does not build the rest of the system around his software implementation of a problem solution, but must build his implementation within the framework of an already existing system.

2. Understanding the Documentation

By choice, a maintenance programmer would prefer to use documentation, instead of going to source code directly, to point him to the segment of code where a program error is or to understand what portions of a program will be impacted by an impending change implementation.

Documentation is essential for software maintenance. If done correctly, it adds significantly to program understanding. Documentation can express what a program does and why. It can reconstruct the intentions of previous programmers and it can anticipate possibilities for future change. Of the many kinds of documentation that can be

created, the most useful for software maintenance is high-level documentation that explains the overall purpose of the program and describes the relationships of the various program components with each other. [Ref. 9:p. 174]

If documentation is not adequate, however, it is better to not have any at all than to have incorrect, imprecise, conflicting, overlapping, or out-of-date documentation. [Ref. 14:p. 67]

Farley [Ref. 15:p. 89] describes what should be included as documentation:

- Product Overview and Summary
- Development, Operating, and Maintenance Environment
- External Interfaces and Data Flow
- Functional Requirements
- Exception Handling
- Early Subsets and Implementation Priorities
- Foreseeable Modifications and Enhancements
- Acceptance Criteria
- Design Hints and Guidelines
- Cross-Reference Index

To this list the following general concepts should be emphasized within a documentation system:

- need to know what the constraints are
- ability to make changes
- serve as a reference tool

- characterize acceptable responses to undesirable events
- document essential details
- list different kinds of changes—those that will not change and those that may change
- document things you cannot figure out for yourself
- need different views and different levels of detail [Ref. 16]

Martin and McClure [Ref. 9:pp. 177 -187] identify four classes of documentation that are needed:

- User documentation—instructions how to use the software
- Operations documentation—instructions how to execute the software
- Program documentation—divided into two parts:

  (A) Source Code—is documentation within itself, used to help understand the internal structures of a program and how programs within a software system interact with one another

  (B) Historical Program Documentation—outlines how a software system has evolved during its development and earlier maintenance phases and is comprised of:

    (1) System Development Journal—includes system development strategies, decision-making strategies, and reasons for selecting a particular design alternative

    (2) Error History—can expect to find future program errors in code segments where heavy error occurrence has occurred in the past

    (3) System Maintenance Journal—how and why a system has changed

3. Understanding the Source Code

Once the maintenance programmer has understood the problem and understands any of the available software documentation, he is still not completely prepared to make source code changes.

Before he can accomplish this task, he must understand the source code that he will modify. Understanding the problem and the documentation should help the maintainer zoom in onto the particular code segment or segments that apply to the specific problem under consideration. Unfortunately, neither problem understanding nor the documentation can directly tell the maintainer what is wrong. The best they can offer is assistance, help towards finding the target segment of code.

Understanding source code is difficult. It typically was not written by the person doing the maintenance. It may not meet the organization's programming style and conventions. Its documentation may, also, be completely out-of-date. Because of the previous reasons, source code suffers a readability problem and it is often difficult to tie the specific problem and a documentation specification to particular code segments.

Program readability can be improved by the use of automated software tools [Ref. 9:p. 366]. Cross-reference listings, symbol tables, automatic flow charters, code reformatting tools, etc. can help change source code into a more readable format.

Martin and McClure [Ref. 9:pp. 364-370] make three other suggestions that can improve program understanding. Their first suggestion is to allow software maintainers the time to develop a top-down understanding of the software system. Second, maintainers should constantly be seeking to improve documentation. Third, maintenance personnel can receive a very complete and in-depth

understanding of the system they are to maintain if they are allowed to participate in program development. Maintainers should participate in software design reviews and coding reviews, and should actively participate in the testing phase. Software maintenance personnel can greatly assist in the development effort because of their past experience and their insistence in helping developers release a more maintainable software product.

4. Modifying the Code

Once the segment or segments of source code that must be modified are identified, it is important not to just blindly go in and change the code. Martin and McClure [Ref. 9:pp. 371-376] specify three steps that should be taken when existing programs are modified: design the program change, alter the code, and minimize side effects.

Martin and McClure [Ref. 9:p. 372] recommend a top-down approach in designing a program change. The entire program should be reviewed at a high level to determine what aspects will be affected. Next, the code portions that will be changed are identified. Finally, the specific change within each module and data structure are specified in complete detail. The program change design must take into account any potential side effects the given change will have on other unchanged segments and the program as a whole. If this is done, then if the code is modified as designed side effects will not occur or at the very least will be minimized.

5. Debug

No one writes perfect code. After the code is modified, it probably will have software bugs in it and must be debugged. The comments that follow apply equally well to the debugging phase after modifying the source code as well as to looking for program errors during a corrective maintenance phase.

Martin and McClure [Ref. 9:p. 382] cite some interesting findings concerning the debugging process.

> First, the inability of experienced programmers to detect even obvious errors is alarming. Second, computer-based debugging by the original programmer appears to be one of the least efficient debugging methods. Third, no single method used alone is very good.

It is hard for programmers to find errors. They often look in the wrong spot. They often have great difficulty in understanding an error's total effect on the program as a whole. Programmers differ greatly in their debugging ability and the number and types of errors they are able to find. [Ref. 9: p. 382]

Group techniques have proved more effective in terms of costs and the number and types of errors found than results achieved using a single programmer. Group techniques include code walk-throughs with several people or simply having two programmers work together to debug a program. [Ref. 17:pp. 129 - 130; 28 -29]

A combination debugging approach that pools different methods and uses more than one person is the preferred debugging

method. Using more than one programmer in the debugging process will also improve programmer education and communication. Software debugging tools may aid the process as well. [Ref. 9: p. 383]

6. Test

After installing a software change, the maintenance programmer must test the modification. He cannot prove that his change is completely correct without doing exhaustive testing, but he will prove that the modification is free of the software bugs he is looking for, that it performs a function, and that it is ready for regression testing and revalidation with the existing software.

7. Perform Regression Testing

Even if unit testing is done correctly, the installed modification cannot be trusted. Regression testing is necessary to ensure that the change does not have a ripple effect on the system as a whole and that the system performs as good as or better than prior to the change. In addition, software must be tested to reaffirm its ability to comply with system specifications, performance requirements, and quality control standards. [Ref. 12:p. 136] The only way to do this is to develop standard revalidation procedures. These standards should closely match the original program validation test cases and test data to allow results from the revalidation effort to be compared with the original test results. Program regression will be obvious when these two test results are compared. [Ref. 9:p. 376]

Revalidation should be done by an independent group separate from the maintenance shop. This independent test organization

should develop standard revalidation procedures for each program and/or system of programs. This group should perform error analyses and complexity profiles and ensure those program areas identified as having high error rates and being highly complex receive heavy testing. The revalidation procedures developed can be greatly aided by software tools like cross-reference systems, test data generators, and file comparison utilities. The most important point is that revalidation procedures must be used. They never should be skipped. [Ref. 9:p. 376]

8. Update Documentation

As previously mentioned it is extremely important to continue to improve the available documentation and keep it up-to-date. All changes to the source code must be documented and a version of the pre-modified code also should be maintained. It is important to remember to update user and operational manuals as well as the system documentation to accurately reflect the software modification. [Ref. 12:p. 154]

V. COMPUTER PROGRAM COMPREHENSION

A. INTRODUCTION

It is not feasible to explore in depth all eight steps of program maintenance. Survey data on how maintenance personnel spend their time shows that the dominant activity is reading and understanding source code and documentation. This activity is obviously the primary focus of the first three steps of software maintenance but it is also an important activity in other steps as well [Ref. 6]. For these reasons, program comprehension will be the feature of this chapter and the focus for the rest of this thesis. First the survey and its results will be explained. Next, a review of the literature written on program comprehension will be presented. Last, the pros and cons of the three program comprehension models described in the literature review section will be addressed.

B. FJELDSTAND AND HAMLEN STUDY

Of the eight steps of performing program maintenance, the steps that dominate are those that are related to reading and understanding. Fjeldstand and Hamlen [Ref. 6] studied how maintenance personnel spend their time. They found the following in their study of 25 IBM installations:

- STUDY REQUEST 18%
- STUDY DOCUMENTATION 6%
- STUDY CODE 23%

- IMPLEMENT 19%
- TEST 28%
- UPDATE DOCUMENTATION 6%

There are a number of important concepts to derive from this study. First, almost half of a maintenance programmer's time is spent reading and understanding what the programmer needs to do. It should be noted that, of the three reading and understanding categories (the first three listed), the programmer spends the most time studying source code. Comparatively, little time is spent actually modifying the code, certainly less than one might expect. Testing takes up a significantly larger portion of a maintenance programmer's time, more so than in development activities. This is not surprising because of the greater need to do regression testing in software maintenance. The maintainer must be totally satisfied that the software change does not impact or degrade the rest of the software system. Another note of interest is that the same amount of time is spent updating documentation as in studying documentation.

Software maintenance is the dominant activity in the software life cycle. Lientz and Swanson [Ref. 18:p. 153] surveyed 487 data-processing organizations and found that both large and small organizations spend on the average 44.4 percent to 53.5 percent of their time on software maintenance. It is equally obvious from the Fjeldstand and Hamlen [Ref. 6] study that reading and understanding are the dominant activities in software maintenance. For this reason, studying

program understanding has the greatest potential to reduce software maintenance costs and software life-cycle costs in general.

Looking at the eight steps in program maintenance points out another reason to study program comprehension. The activities of implementation, testing, regression testing, and updating of the documentation tend to be standardized. Most organizations specify to their maintenance personnel how these activities will be completed. How a maintenance programmer understands the problem, understands documentation, understands source code, and debugs are informal and highly individualized.

Understanding the problem and debugging are activities involved in all programming. Certainly, debugging is universal to all programming; however, understanding the problem in development work is different from understanding in maintenance. This point has already been made above. Activities related to reading and understanding are the least well understood. This, together with the fact that they represent the core activities of software maintenance, points out why the study and analysis of these activities are so important. The next section provides an overview of program comprehension models.

C. COMPUTER PROGRAM COMPREHENSION MODELS

Not much has been written on program comprehension, but three models have been proposed. The subsequent sub-sections detail each one of the three models described in the literature.

1. Syntactic/Semantic Model

The syntactic/semantic model is more comprehensive than the other models. This model proposed by Schneiderman and Mayer [Ref. 19] attempts to describe an all encompassing view of the programmer's task. Besides defining program comprehension, their model additionally incorporates the following programmer behaviors: program composition, debugging, modification, and the learning of new programming skills.

The backbone of their model revolves around two basic themes. The first is the role of three different levels of memory: short-term memory, long-term memory, and working memory. The second is the difference between syntactic and semantic knowledge.

Schneiderman and Mayer [Ref. 19] describe short-term memory as the means through which information from the outside world enters the cognitive process. Little, if any, processing is done on information at this memory level. In contrast, long-term memory contains information that has been fully processed and organized. It represents an unlimited store of knowledge that is available for recall. Working memory is a bridge between short-term memory and long-term memory. It is the epicenter of the problem solving process. It pools information that is fed into the human cognitive system via short-term memory with relevant, associated knowledge that it calls from long-term memory. The result of this mixing in working memory is the genesis of a problem solution that can either be produced and forgotten or stored in long-term memory for future reference.

Because of the nature in which each of the three levels of memory interact on information, Schneiderman and Mayer [Ref. 19] have in effect produced a broad information-processing model. [Ref. 19:p. 220]

Their view of syntactic and semantic knowledge is aligned with a computer scientist's version of these terms as they apply to a programming language. "The syntax of a language is the way that words and symbols are combined to form the statements and expressions." [Ref. 20:p. 89] Semantics is "the meaning of well-formed expressions." [Ref. 21:p. 2-12] Both, according to the syntactic/semantic model, are stored in long-term memory. [Ref. 19:p. 221]

Particularly illuminating is the difference in ease of learning syntactic knowledge and semantic knowledge across programming languages. Although it is hard to learn a first programming language, learning a second programming language is easy provided the two languages share similar semantic structure (i.e., both are structured languages). If they do not, then learning the semantics of the second language actually can be more difficult than learning the first language. Syntax knowledge is just the opposite. The closer the syntax of two languages is the easier it is for a programmer to confuse and incorrectly substitute one language's syntax for another. The further apart the two languages are, the easier it is for a programmer to keep each language's syntax rules separate. [Ref. 19:p. 221]

The Schneiderman and Mayer [Ref. 19] view of program comprehension may be termed a pure, bottom-up approach. It also

relies heavily on George Miller's "process of chunking" [Ref. 22:pp. 81-97] that was used in describing limits to processing information.

In the syntactic/semantic model, the initial step a programmer takes in understanding a program is to read the source code. The source code is read first for syntactic understanding. Syntax knowledge is used to provide a link to develop a higher-level semantic understanding of what the program functionally does. Syntax is not learned line by line but is learned in pieces. These pieces of knowledge are "chunked" [Ref. 22] together to form bigger pieces of understanding until the entire program is comprehended. Naturally, this "chunking process" is aided by the use of modular program design and structured programming languages. [Ref. 19:pp. 224 - 225]

Schneiderman and Mayer [Ref. 19] emphasize that, although low-level syntax details may not be fully understood, it is still possible to develop a high-level comprehension of the program. In addition, it is also possible to fully understand a program on a low level, yet never achieve a full, high-level understanding of the program as a whole.

2. Hypothesis Model

While the syntactic/semantic model defines a broad information-processing theory, Brooks' [Ref. 23] hypothesis model focuses on the more narrow process of program comprehension. The basic idea of this theory is that when a program is written it is constructed from a series of mappings from a problem domain to a program domain. A program is comprehended by creating a hypothesis that bridges the gaps between domains. Specifically, a hypothesis, whether on a high

or low level, will link the problem domain and all intermediate domains with the program domain. [Refs. 23, 24, and 25]

The process of creating a hypothesis is iterative. Brooks [Ref. 23] specifically states that, as soon as a programmer has any knowledge about a program, he makes an initial high-level hypothesis about how the program works. The programmer tries to gain confirmation that his hypothesis is correct by examining source code or other related documentation in an attempt to find a match. If he does not find an exact match he will refine his hypothesis or change it to create a closer link with the code and documentation. It is important to note that hypothesis generation is done in a top-down fashion, achieving greater refinement and elaboration. [Ref. 23:pp. 544-550]

Hypothesis generation is an on-going process. It continues until the programmer feels the successive versions of the hypotheses have been fine-tuned enough to be relatively close to the actual program code or documentation. Although the concept of "relatively close" is not well defined, it occurs when actual data structures and operations defined within the hypotheses can be either found or closely associated with similar features and details in the program code or documentation. Brooks gives a special definition to the code line related to these features or operations. He defines them as beacons. [Ref. 23:p. 548; Ref. 25:p. 127]

Beacons play a key role in further refinement and specification of the evolving hypotheses. In particular, beacons are tied to lines of code. Beacons become the means through which lines of code are

bound to the hypotheses. Of significance, the possible existence of program beacons has been strengthened through experimental results. Wiedenbeck's [Ref. 26] research supported the theory that beacons provide comprehension focal points for experienced programmers. [Ref. 26]

Brooks' [Ref. 23] hypothesis model also implies specific documentation needs. Because initial hypotheses are general and broad in nature, high-level documentation, such as design descriptions and user's manuals, must exist. In other words, the generation of initial hypotheses may be limited by using source code alone. Although Brooks [Ref. 23] points out that redundant documentation is not desirable, a certain level of documentation at all levels of hypotheses generation must exist, because it is documentation which contains information that will allow binding between domains to occur. [Ref. 23:pp. 551-552]

In Brooks' [Ref. 23] comprehension of computer programs theory, he stressed three distinct concepts that defined why programmers exhibit different levels of ability in comprehending any one given program. Programmers differ in the degree of programming ability they have, in the amount of specific problem domain knowledge they have available to apply to hypotheses generation, and in the variety of comprehension strategies they may employ. The first can be improved with more experience and training. The second may be improved by documentation that clearly describes the rationale behind a program's specification. The third may be aided by merely alerting

and educating programmers about the strategies available for their use. [Ref. 23:pp. 553-554]

3. Slice Method

The third and final model, the slice method, is considered a debugging method [Ref. 27:p. 381]. Although debugging and program comprehension are considered two distinct tasks, there is a commonality between them. Namely, before a programmer starts to debug, he already understands to some degree, or at least should, the program he is trying to correct. The slice method gives an explanation of how much a programmer needs to understand of the program he is debugging.

Regardless of methodology (function driven versus data driven) or implementation (top-down as opposed to bottom-up), a program designer or writer is trying to decrease the amount of information he must comprehend at one time. The same is true for a program maintainer. The need exists to divide a large program into parts whose function and scope of action are easier to conceptualize.

Slicing performs this decomposition. It is a means to decompose already-written programs into subsets of program behavior. The idea is the programmer is interested only in looking at a specific behavior of a program at one time, rather than the program as a whole. By applying the correct slicing criteria, all code but what is irrelevant to the specific behavior can be stripped away. Although all irrelevant code is removed from view, what remains is code that is still capable of demonstrating the desired subset behavior of the

original program. The slice is generally composed of noncontiguous fragments of the code. [Ref. 28:p. 439; Ref. 29]

Obviously, there is more than one way to decompose a program. Depending on the slice criteria, what results is essentially a different view of the program. Each view offers a different context in which to understand the program. Some views will be better for converting certain errors then others. In addition, specific views will also be better to suggest what the error is.

Ignoring code that does not apply to what you are trying to change is not limited to the debugging task. It applies equally well to software maintenance. Except in maintenance, the need is to ignore all code but the code portion that must be improved or replaced. [Ref. 28:pp. 447-448; Ref. 29]

D. PROS AND CONS

This section attempts to highlight the strong and weak points of each of the three computer program comprehension models.

1. Syntactic/Semantic Model

Schneiderman and Mayer [Ref. 19] are correct in their analysis of low-level and high-level understanding of a program. It is possible to fully understand a program on a low level yet never achieve a full, high-level understanding of the program as a whole. For a maintenance programmer, it is much more disastrous to err by not understanding a program on a high level than on a low level, because of the global consequences any modification made may have on the program as a whole. If the maintainer does not understand the program at a

high level, how can he even hope to appreciate the effect of the changes made across the program's entire scope?

In addition, there is a keen distinction between program comprehension and program composition. As already described, in Schneiderman's and Mayer's view [Ref. 19], comprehension is bottom-up. It moves from syntactic to semantic knowledge. Program composition on the other hand is top-down. The programmer fully solves the programming problem on a semantic level only. He employs his syntactic knowledge in a straightforward, almost mechanical, manner when he writes code. The task is easier when you can separate the use of syntactic knowledge and semantic knowledge, as in writing a program. This is in contrast to when you understand a program where you are always using a mix of the two. [Ref. 19:pp. 223-225]

This model's description of the chunking process is also a positive factor. Programmers do chunk together closely related portions of code.

The emphasis on a bottom-up approach is a negative aspect. In maintenance, the need is to initially start with a top-down approach. Bottom-up is typically used only after the maintainer has identified the code segment that has to be changed. Once identified, the maintainer may take a bottom-up approach to understand precisely what is going on in the code step-by-step.

The Syntactic/Semantic Model also errs in making the reading of source code the initial step in program comprehension. Reading source code as a first step is not what we want to do. It may be

what always has been done only because documentation has been so poor (i.e., incomplete, out-of-date, conflicting, etc.).

2. Hypothesis Model

The hypothesis model allows for a top-down approach. It matches a maintainer's need to have a high-level view of what the user needs (operational sites) and what the software system does.

It has levels and steps. It accounts for different degrees of program comprehension as the degree of experience and exposure increases.

The hypothesis model describes the notion of iterative understanding. Although it is desirable for a program maintainer to fully understand the code they are maintaining, certain situations may occur to prevent this from happening. The programmer responsible for a particular section of code with a bug in it may be out of town. Another maintainer may be able to learn enough about the code to make the change. His level of understanding of the given code segment changes through the maintenance process. The maintainer may have walked in with only a high-level overview understanding of the target problem code, but by the time he has corrected the problem he will have gained greater knowledge of the problem code segments. Even so, the knowledge he will have gained will not be as great as that of the assigned maintainer.

An additional advantage of the hypothesis model is its concept of beacons. Beacons are means of abstraction. They are a way for a programmer to give a name to a code section. When a programmer

writes code or is in a debugging phase, he does not need to re-read each line of code to know what is going on. He sections it, or "chunks" it, and ties a name to the section. The name is the beacon. The beacon can be a variable name or a short phrase explaining the function of the code block.

3. Slice Model

The slice model is not a model of comprehension but a method to help improve program comprehension. It gives a variety of views of a program and can be used either in a top-down or bottom-up fashion. It provides a means to zoom in on relevant lines of code and to ignore others. For this reason, it is a useful means to reduce program complexity. The slice method allows source code lines that are related to each other but texturally disjoint to be viewed together.

The slice method offers one distinct advantage over the other two models. The degree of program comprehension required to use the slice method is significantly less. In fact, it is extremely well suited to being used in situations when the program to be maintained is large and unfamiliar to the maintenance programmer. [Ref. 28:p. 439; Ref. 29]

4. Summary

Of the three, Brooks' hypothesis model is the most closely aligned with the software maintenance task in general. Its top-down approach and iterative understanding most closely explain what the maintainer must do. As such, Brooks' theory will be used as the model of program comprehension in the rest of this thesis.

VI. WHAT BROOKS' THEORY PREDICTS

A. INTRODUCTION

If Brooks' theory is accepted as a reasonable model of how a programmer tries to understand a program and its associated documentation, then what does Brooks' theory predict will be the potential problems of a software maintenance organization? The rest of this chapter will explore the implications of Brooks' Hypothesis Theory as it applies to this question.

B. PREDICTED POTENTIAL PROBLEMS

Brooks emphasizes the need to develop different levels of understanding and to develop this understanding in an iterative, top-down fashion. If this is true, one of the problems a maintenance organization will face is how to package knowledge at discrete levels. Brooks' theory predicts that programmers look only at documentation that corresponds to their current level of understanding rather than looking at all available documentation. Brooks' theory also predicts that programmers gain knowledge about a program by first achieving a big-picture, top-level view. They attempt to understand a program from a general, functional level, before they understand specific lines of code. Programmers will understand what problem a particular program is trying to solve before understanding how the program code solves the problem. A maintenance organization thus faces a problem in deciding what types and forms of documentation and software tools are needed

to aid the development of iterative understanding achieved in a top-down fashion. In addition, Brooks' theory suggests that understanding the original problem the software design organization was trying to solve, the specifications they were working with, and the why they choose certain design decisions will be critical information for a maintenance organization. In fact, this information must be gained and well understood before other knowledge can be adequately achieved.

Brooks further specifies a programmer's ability to understand a program in a top-down, iterative fashion as hypothesis building. To recap what has already been expressed in the previous chapter, Brooks' theory predicts that a programmer's understanding must move through a series of domains, from problem domain, to specifications domain, to database domain, to application domain, to program (computational structures) domain. Thus, documentation and software tools chosen must help the mapping from one domain to the next. They also must allow a programmer to generate hypotheses that answer what, how, and why questions about the interrelationships between the problem domain and the program domain. Tools that help hypothesis generation cannot be restrictive but must allow hypotheses about how the program works to evolve as new information and understanding are gained.

Another problem that Brooks' theory raises is the issue of programmer variability. Because a programmer's level of understanding changes with time, experience, and exposure, all programmers within

an organization will not understand the software system they are maintaining to the same level or to the same degree. Numerous studies have been completed that indicate programmer ability varies as much as 26:1, 10:1, and 5:1 [Ref. 30:p. 19]. How does an organization deal and cope with a variance as wide as this in programmer ability?

Since programmers vary so widely in ability, this implies that each programmer will have a different view and understanding of the program he is maintaining. A maintenance organization cannot run efficiently, let alone survive, the chaos that would reign if each maintenance programmer made changes to source code according to his own hypothesis of how the program currently works and how it should work. A significant problem for the maintenance organization will be how to develop, maintain, and enforce one common program view for the organization. A common understanding of both the desired application behavior and the computational model to be applied in developing the system is necessary.

If the problem of different programmer views were not enough, the views of users, management, and the maintenance organization are all also widely different. This occurs for the same reason as it does among programmers. Users, management, and the maintenance organization have different levels of experience and expertise. Each has its own different focus on the role, function, and meaning of the software system. Each comes to its own specific view from its own unique perspective. How does each of these groups communicate its different viewpoint to the others? This question is of greater concern

to the maintenance organization than to the users and management because the maintenance organization must keep the user and management happy at all times. For the maintenance organization, the communication problem not only concerns how it should or how it can communicate its view of the software system to users and management, but also how to understand what the user and management are trying to say from the perspective of their viewpoint and level of understanding.

Brooks' theory predicts that the way to deal with the variance in views between users, management, and the maintenance organization is to develop a "theory of the field." The theory of the field contains all information about the problem domain, the specification domain, the database domain, the application domain, the program domain, and all the ties and interrelationships between each of these domains. The theory of the field will allow the Software Support Activity to structure the knowledge about the field (the domains and their interrelationships) in order to manipulate, preserve, teach, and re-capture it. Developing the theory of the field would be like developing a curriculum or writing a book or a seminar. The theory is the joint expertise that the Software Support Activity provides users and management. The theory of the field that the Software Support Activity develops should not be haphazard, but planned. The Software Support Activity must know what it stands for, what its mission is.

Other authors support Brooks' theory. The most notable are Martin and McClure, MacLennan, and Curtis, Krasner, Shen, and Iscoe.

C. MARTIN AND McCLURE

As already described, Martin and McClure [Ref. 9] stress the importance of high-level documentation for software maintenance [Ref. 9:p. 174], the need for different levels of documentation [Ref. 9:pp. 180-185; pp. 366-367], and the need for maintainers to get involved early in the life cycle [Ref. 9:p. 367]. This last issue is presented by Martin and McClure [Ref. 9] as a means to allow a maintainer to learn the background of a software system, the problem domain and specifications, and to understand why certain design decisions were made and implemented.

D. MACLENNAN

MacLennan [Ref. 31] also supports the need for different levels of documentation and understanding. MacLennan defines fifteen requirements of a computerized software development environment. Of the fifteen, seven of his specifications apply directly to Brooks' theory. They are as follows.

1. Simulated World

A software environment should be able to represent the entire software life cycle. To do so, an environment must be able to represent and manipulate within the computer a large number of objects, both current and future, and their interrelationships. Some

examples of objects are Data Flow Diagrams, code, people, specifications, and computer resources. In order for a software environment to model the software life cycle, it must provide a simulated world. For an object that is concrete it may not be possible to represent that object directly in the computer. The object may have to be simulated. The same may be true of the large number of relationships between objects that may be represented. If a simulated world were achievable, it would allow mapping from one domain to another and allow transformations among and between domains. [Ref. 31: p. 1-2]

2. Persistence

A large software project typically takes years to develop. As a consequence, the objects and relationships within an environment do not go away. They must be stored for the life of the project, through implementation and maintenance, until the software is no longer used. If this information were maintained for the life of a project, then a maintainer would have access to what was the original problem that needed to be solved and what were the original specifications. [Ref. 31:p. 1-2]

3. Uniformity

Despite the fact that objects and their relationships vary widely, they must be treated and manipulated in a uniform way. If you are to map from one domain to the next, then a uniform way must exist to do the translation between domains. [Ref. 31:p. 1-2]

4. Flexibility

Software projects evolve. They change with time. As a result, the objects and their relationships also must change. A software environment must be flexible enough to allow these changes to occur almost naturally with little or no impact on users. Not only do software projects evolve, but so also do the hypotheses programmers make about programs. Flexibility to change these views easily is highly desirable. [Ref. 31:p. 1-3]

5. Alternate Representations

An object may have more than one visual representation. Programmers view objects in different ways based on past experience and exposure. Their different views should be supported. [Ref. 31:p. 1-3]

6. Multiple Views

When an object does have alternate representations, if one of its views is changed, then you want all visual versions of the object to be updated relative to the change. All views must be made and remain consistent. The consistency of views and understanding within an organization has already been stressed. [Ref. 31:p. 1-3]

7. History

The computer can provide immeasurable assistance by keeping track of a project's history. What should be recorded is changes to specifications, personnel, design decisions, code, goals, etc. What must also be recorded is the known cause of the change. [Ref. 31:p. 1-3]

Project information and programs change. Often what has changed in the past, will be changed again in the future. Knowing the reason why something was changed, helps personnel to design and implement better solutions.

E. CURTIS, KRASNER, SHEN, ISCOE

Curtis, et al. [Ref. 32] have produced experimental results that supports a large portion of Brooks' theory. Their survey of nineteen projects from nine companies yielded results that not only supported the notion of programmer variability and their resultant differences in degrees of knowledge [Ref. 32:p. 97], but also stressed the importance of treating the development and maintenance of large software systems as largely a learning and communication process. [Ref. 32:p. 102] In fact, Curtis, et al. [Ref. 32] propose that the processes of learning, technical communication, negotiation, and customer interaction are among the most crucial to any project's success. [Ref. 32:p. 103]

They found in their studies that software development contains a large commitment of time dedicated to learning. The knowledge required to develop the system absorbed most of the project team's time during the early stages of the project, because "much of what occurs during design is not designing, but learning required in order to design successfully." [Ref. 32:p. 100]

This finding applies equally well to software maintenance. Design is involved to some degree regardless of which of the three categories of maintenance is being performed (corrective, adaptive, perfective). Considering that 55 percent of all maintenance done is perfective

(maintenance done to enhance the capability and performance of the system and to improve the documentation and software) just adds to the claim that learning consumes a significant portion of maintenance time. For the reader's interest, corrective maintenance consumes 20 percent of the total time spent on maintenance and adaptive consumes 25 percent [Ref. 18:p. 68]. But perhaps the more telling finding of the Curtis, et al. survey [Ref.32] is the documentation of the tremendous amount of time most projects spend rediscovering information that had been generated by the users and originally held by the design organization [Ref.32:p. 101]. If design organizations spend a lot of time on this task, then maintenance organizations will spend even more time because of the time difference inherent between a project's inception and its maintenance phase.

Technical communication and negotiation become imperative to ensure that organization members share the same model or view of how the system should operate [Ref.32:p. 100]. Curtis, et al. [Ref.32] presented an idealized scenario of how an organization resolves differences between each team member's individualized view of the project. Although all members start out with their own mental model of what should be done, group members start to form coalitions with other members who share similar views. The coalitions are formed to argue their group member points of view. The final stage of this negotiation process is marked by the resolution of all differences between the coalitions and the development of a team consensus. [Ref.32:p. 101]

Curtis, et al. [Ref.32] observed, however, that although the formation of multiple coalitions was desirable in order to gain the benefits of alternative views, in practice this rarely happened. A single individual or group formed a dominant coalition that took control of the project and dictated how the system should operate. "In fact, in some cases the members of the dominant coalition even acknowledged that they had formed a steamrolling group to move the project in the direction they believed it should go." [Ref. 32:p. 100]

Their finding, because of its simplicity, tends to downplay the vast amount of communication found to be necessary to ensure that all members of an organization share the same understanding of the system. The amount of communication needed is so great that Curtis, et al. [Ref. 32] make two recommendations to deal with this problem. One is the recommendation to develop formal organizational structures that will help communication flow horizontally across an organization rather then just vertically upward. The second is to augment informal communication methods with better "coordination tools." [Ref. 32:p. 103]

Curtis, et al. [Ref. 32] fully support Brooks' theory that programmers and users share different domains of knowledge [Ref. 32:p. 96] and the degree of difference, if great, can adversely impact the future of the project [Ref. 32:p. 99]. Based on their survey, Curtis, et al. [Ref. 32] recommended that one organization source be identified to clarify user requirements to the organization [Ref. 32:p. 102].

It is already obvious that the Curtis, et al. [Ref. 32] finding described above provides the Brooks' theory significant support. But what perhaps sheds more light on the Brooks' theory is Curtis, et al.'s [Ref. 32] identification and definition of an organization expert that they term the "super-conceptualizer." [Ref. 32:p.99] Communication and education of all organization members to ensure that they share the same common model of how the software system should operate is the most significant function of the super-conceptualizer [Ref. 32:p. 102]. The super-conceptualizer is the person or a small group of individuals who are "the keepers of the product vision." [Ref.32:p. 99] They are the application experts who are skilled at communicating their technical vision. A super-conceptualizer's unique vision is the ability to "map between behavior expected of the application system and the computational structures required to create this behavior." [Ref. 32:p. 99] This is done despite the finding that super-conceptualizers often admitted they were not good programmers. [Ref. 32:p. 99]

Super-conceptualizers are further categorized by Curtis, et al. [Ref. 32:p. 99] as being

> ...dedicated to and consumed with the technical performance of the project. In so doing, they frequently became the primary source of coordination on the project and assumed, without formal recognition, many management responsibilities for ensuring technical progress.

F. OTHER PROBLEMS

This is not to say these are the only problems the organization will face. There are others. Some of the more important ones include how to manage inevitable changes in requirements, how best to deal with the overwhelming complexity of large programs, how to accomplish version control and configuration management of multiple copies of the operational system, how to protect the system so only key personnel can make changes, and how to offset or counter the efficiency versus maintainability dilemma. The Software Support Activity has two goals. One is to make source code more efficient and the second is to write code that is easy to maintain. Efficient code is not easy to understand nor is it easy to modify. For these reasons, efficient code is the antithesis of code that is easy to maintain.

G. IMPLICATIONS FOR THE SOFTWARE SUPPORT ACTIVITY

Although there are other issues that a software maintenance organization must face, the premise of Brooks' theory and the supporting work of Martin and McClure, MacLennan, and Curtis, et al. is that communication and learning issues are the most critical for a software maintenance organization. Although these critical concerns are supported by several authors, these issues cannot be proven to be significant for the Software Support Activity at this time. What would be valuable is to plan to survey the Software Support Activity one year after it assumes software maintenance responsibilities to determine what the Software Support Activity considers its most difficult problems. The problems the Software Support Activity actually

encountered could then be compared to the problems predicted in this thesis. Regardless, considering the overwhelming evidence that identifies communication and learning as the core, critical issues for any organization, it is prudent to identify and plan methods and measures to help reduce their impact. The next chapter will outline some ideas and plans to help reduce any negative influence the lack of proper communication and learning may have for the Software Support Activity.

VII. IMPLICATIONS FOR MANAGEMENT AND TRAINING

A. INTRODUCTION

Brooks' theory does not say anything about software tools. The theory is, however, specific about documentation. According to the theory, documentation must present information in a top-down fashion and provide levels of understanding. The premise of this thesis is to carry the same ideas Brooks' theory predicts are important for documentation and understanding of programs to the selection of software tools. Based on what Brooks' theory says is important, how do the tools selected for the Software Support Activity rate?

B. RATING THE SOFTWARE SUPPORT ACTIVITY TOOLS SET

In Chapter III, the Software Support Activity tool set received an outstanding rating as an environment. The level of integration of the Software Support Activity's tool set is one of the dominant reasons why it received such a high mark. Brooks' theory reinforces why an integrated environment is desirable. An integrated environment allows mapping or translations between software tools. Since the Software Support Activity's tool set is also an example of a non-modal environment, the translation of needed information between each tool almost seems transparent because tools can be used within other tools.

Of all the requirements of Buxton's and Druffels' environment standards [Ref. 10], only one aspect was not fully met. This was the

environment requirement to support the entire software life cycle. None of the tools in the Software Support Activity tool sets automate the front-end of the life cycle. None of the tools helps with analysis and design. None of the tools deals with problem definition or linking specifications with lines of code. All the tools emphasize the programming task. This is not surprising. The whole issue of environments, even a definition of what an environment means, is still hotly debated within the software community. "Software support environments are still too incompletely understood to specify precisely." [Ref. 33:p. 42] It also is not known whether environments will actually help productivity. The software community just thinks environments will. Boehm, et al. [Ref. 33] have produced the only study results; the availability of software tools improved productivity by 15.6 percent [Ref. 33:p. 41].

Environments are in their infancy. They have been talked about in the literature for only the past couple of years, but it "takes typically 17 years (±2) from concept inception to commercialization for an automated software technique" [Ref. 1:p. 23] to become widely accepted.

The same is not true for software tools in general. The notion of software tools is a well-known and accepted concept. Literally thousands of tools exist and the vast majority are programming aids. Programmers are the people who have developed software tools. They have developed tools that help automate tasks that they, the programmers, deal with on a day-to-day basis. Their view has not

necessarily been one of implementing an integrated tool set nor developing a tool set that supports the entire software life cycle or mappings from one knowledge domain to the next. Based on a programmer's narrow, specific view, the software tools available are largely bottom-up tools. The "top" of the software engineering process and its automation is missing [Ref.1:p.116].

The same can be said for most of the tools in the Software Support Activity tool set, but there is a difference. It is unusual for a set of tools to be as integrated as the VAXset. At present, there are two directions an organization can go in selecting software tools. It can select many different tools, and there are many good software tools available, or it can select a small, integrated set.

The advantage to selecting a large set of tools that are not integrated is a new useful tool can be added at any time. The disadvantage is no one will learn how to use all the tools. For a small, integrated tool set, the learning process is easier and as a consequence all the tools will be used more effectively. The disadvantage is that when new software tools become available the organization must fully evaluate all the costs involved in adding the tool. The translation and re-education processes required to add a new tool, especially one not fitting the present environment, could be expensive. The cost of adding the new tool may offset any advantage gained by incorporating the new tool, regardless of how valuable it is.

For the Software Support Activity, the right choice has been made. Due to the experience level of its personnel and the expected

turnover, the Software Support Activity will do better with a small set of powerful tools, and that is exactly what it has. As a further argument, VAXset capabilities are impressive when they are compared with other industry products.

C. HOW SHOULD THE ORGANIZATION RESPOND TO THE LACK OF TOOLS?

Although the Software Support Activity's tool set is the right choice with respect to what is currently, commercially available, it does have limitations.

The limitations are those issues identified in Brooks' theory as potential problem areas for an organization. No specific tools have been developed to counter each of the six identified issues. This section will present some suggested approaches to help alleviate the problems this lack may produce for the Software Support Activity.

1. How to Develop Different Levels of Understanding? How to Develop Understanding in a Top-down Fashion?

The real issue is what documentation should the Software Support Activity select for its use and how should the selected documentation be kept up-to-date. According to Brooks' theory, only documentation that provides different levels of understanding and develops understanding in a top-down fashion should be selected.

In order to meet these goals, documentation should be available on-line. If this were possible, then the programmer would not need to look at all the documentation at one time, but look only at

documentation that is related to the code section he is currently working on.

Studies have shown programmers tend to use only program documentation that is available on-line. It is also the only form of documentation that programmers typically keep up-to-date. [Ref. 34:pp. 100-101]

The ultimate goal of all system documentation is to have it generated automatically.

By choice, all system documentation should be available on-line. Documentation should be included that explains system inputs and outputs, methods and algorithms used, error recovery procedures, all parameter ranges, and default conditions. In addition, the System Requirements, Functional Specifications, all design documents, the Test Plan, test cases, test data, anticipated test results, and User Manuals should all be available upon demand. [Ref. 12:p.55]

What if on-line documentation cannot be achieved? The only possible solution is to consider the use of a document preparation system to re-document the delivered documentation to meet the Software Support Activity's needs. The documentation preparation system must include the ability to organize or index the information the documentation holds, allow the development of cross-reference tools within the new documentation created, and enable the generation of a glossary.

The software tools used must also provide levels of understanding and a top-down organization. As previously discussed, the

high level of integration demonstrated by the Software Support Activity tool set helps provide the required level of understanding. The tools within the Software Support Activity tool set in general do not provide a good top-down presentation of the software system. In fact, they may be largely classified as bottom-up because they deal directly at the code level. An exception to this general rule is the Common Data Dictionary. It is not surprising that this tool was specifically requested to be added to the Software Support Activity tool capabilities because it provides a top-level view to the software system.

The Common Data Dictionary contains all data definitions used within a software system. It knows within which modules, programs, or tools each of the data names are defined. As a result, it provides a higher level view of the software system than looking at code directly. The Common Data Dictionary has other desirable features. It controls access to all data definitions. As a consequence, it will reduce redundancy and inconsistencies between data definitions. Its control will prevent a programmer from creating a second name for a previously created data definition. When a data definition needs to be changed, it must be changed only in one location in the Common Data Dictionary. The Common Data Dictionary will ensure the data definitions are changed in each application program. The Common Data Dictionary will also help the programmer locate the correct definition to use in an application program. [Ref. 35:p. 3-2; Ref. 36:p. 1-29]

The Source Code Analyzer is another tool that provides a high-level view. It was not procured as part of the original Software

Support Activity tool set because it had not been commercially released. It became available to the public in April 1987. It would be an excellent tool to add to the Activity's tool set because of its support of the concepts of providing understanding in levels and in a top-down fashion. The Source Code Analyzer provides static program analysis, cross-referencing, and navigation through source code. The Source Code Analyzer can be used directly from the VAX Language-Sensitive Editor. In other words, it is fully integrated and compatible with the rest of the VAXset tools. More detailed information about the Source Code Analyzer may be found in Appendix D.

2. Understanding the Problem/Specification/Documentation

No new solutions can be presented in this section. The solutions of the previous section—re-documentation, on-line documentation, and a document preparation system—can be equally applied to the issues of understanding the original problem the software system was to solve, the specifications generated, and to the understanding of documentation in general.

What this section re-emphasizes is that the front-end of the software life cycle is the least understood of all the phases. Organizations have the greatest difficulty in representing knowledge of these processes for later transfer to the other phases of the life cycle or to a maintenance organization. For the Software Support Activity, the problem of representing this knowledge and transferring it for later use is compounded because of the high turnover rate of its programmers (three-year tour lengths). The issue here is how to represent

knowledge of these processes and how to make this knowledge persist through the lifetime of the software to be maintained. Present software tools cannot fully automate these requirements. What we are trying to support is nonprogramming activities. Problem definition, feasibility studies, analysis, and system design do not produce any code and often may not even be performed by programmers. What is created out of each of these phases is documentation. "Studies indicate that about two-thirds of the time spent on a large software project results in documentation as its direct product, and only one-third results in code as its direct product." [Ref. 33:p. 32] "Even in the coding phase, peripheral activities—such as the generation of unit test plans, memos, and reports—consume a significant percentage of a programmer's time." [Ref. 33:p. 32] What is being implied is that office automation tools like word processing, forms management, calendar management, spreadsheet, etc. must be integrated into the software environment.

3. Help in Mapping From One Domain to the Next

This is an extremely hard issue. If it were possible to map from the problem definition domain, to specifications, directly to code, then Software Engineering would of achieved its ultimate goal. No one would have to worry about software tools or documentation because source code would be produced automatically. This capability is not currently available, nor may it ever be. It certainly will not happen until the processes encapsulated within each of the domains and their interrelationships are better understood. [Ref. 37]

The only automatic mapping available is the mapping from a program language to its executable source code. This mapping is accomplished by a compiler or an interpreter.

Headway is being made in other areas. Software tools are beginning to emerge that help the translation process from one phase of the life cycle to the next. ProMod is a case in point.

ProMod is marketed as a software development tool. Written in C, it runs either on an IBM-PC or a VAX 11/780 running under VMS. What it does is tie the phases of requirements analysis, structured analysis, and program design together. It does so by carrying information forward from one step to the next.

ProMod is an impressive tool. Its greatest deficit is that it is locked into specific methodologies. Most notably, the Yourdan/Demarco structured analysis methodology, which is a good methodology but may not match every organization's mindset. Despite this particular drawback, ProMod offers a number of desirable features:

- Integration of data flow diagrams, data dictionary, mini-specifications, interface definitions, function call, and data scoping
- Ability to make global changes
- Interactive batch mode graphics and text editor
- Standardized documentation facility
- Consistency and completeness checker
- MIL SPEC 2167 support
- Archive for maintenance

- Generation of pseudo-code templates (pseudo-code shells are provided but the functionality must be provided by the programmer.)

ProMod's power comes from its ability to keep track of all the necessary minute details, their interrelationships, and the ability to trace all key requirements from their current status to the moment of their initiation.

It is not suggested that the Software Support Activity go out and buy ProMod, although ProMod should receive a more critical review. What ProMod represents is the initial cut of software tools that are attempting to automate the front-end of the life cycle. The software industry is just beginning to recognize the critical need in this area. Vendors will be doing their utmost to fill the critical void with their own solution to the problem.

4. Dealing with Programmer Variability

This will be a constant issue for the Software Support Activity. The Software Support Activity will always have a mixture of novices and experts. The need to develop levels of understanding within documentation and software tools has already been adequately addressed. Software Support Activity training and education must meet the needs of both novices and experts.

5. How to Develop and Enforce the Organization's One Common View/Model of the System Being Maintained?

Three sub-issues are involved in this category. They are: (a) how to help the process of learning, (b) how to help improve technical communication and negotiation within the organization, and (c) how

to improve the likelihood of developing super-conceptualizers. Each sub-issue is covered as a separate sub-section of this section.

a. How to Help the Process of Learning

In most organizations, early training of employees is limited and isolated to the specific activities employees are hired to do. Employees, typically, are not given the big picture of how their job fits in the large scheme of things. Software Support Activity training must be different. Software Support Activity personnel must be taught how to transfer and translate user concerns into programmer concerns. The early training of all personnel must emphasize that the theme for the Software Support Activity is service. One way to do this is to establish the following exercise as part of every Software Support Activity member's early training. The exercise would graphically demonstrate the mechanism of how a user requirement is accepted by the Software Support Activity and how it is propagated through the organization and back out to the user as a code, manual, or operational change. Everyone, regardless of department, would trace the steps required to take a user requirement through the organization to the action office and back out again. The training should also emphasize how Software Support Activity performance is measured—a quantification of: (1) How long does it take for the user problem to be understood? (2) How long does it take for the requirement to filter through the organization to the action office? (3) How long does the action office take? (4) How long does the quality assurance group take to certify a good fix has been made? (5) How long does the fix take to

reach the field? (6) How does the user rate the fix once he receives it? Each person's role in this scheme must also be identified and stressed.

What will develop out of this particular exercise is an understanding for everyone of what their job is within the organization, what the role of the Software Support Activity is, and the development of a corporate culture. If done right, the exercise would have a profound effect on how Software Support Activity personnel think about themselves and how they describe the role of the Software Support Activity to people outside the organization.

b. How to Help Improve Technical Communication and Negotiation Within an Organization

The real issue is how to help develop horizontal communication in the organization. A number of things can be done. One is to develop within the organization a theory of the field. In order to help horizontal communication, everyone should have the same, or at least comparable models of the software system. The Software Support Activity should help develop a local vocabulary of technical concepts and their meanings that everyone should use in the same uniform way. A technical library must be created to house the books and papers that support the local vocabulary and view.

Networking of maintenance workstations is a real plus in aiding informal, horizontal communication. Software maintainers need to share and show their work. If they are having problems with a particular segment of code having a co-worker take a look and

providing that look within the computer environment, i.e., via electronic mail or file transfer, is a keen advantage. In addition, having maintainers on the same network will ease the problem of them working together on different parts and on different versions of a large project. [Ref. 38:p. 236]

Walk-throughs are an outstanding mechanism to help the flow of horizontal communication. The Software Support Activity should consider conducting walk-throughs even on code that is not presently being modified, just to help the programmers keep fluent in their responsible program area. These "educational walk-throughs" would serve the dual purpose of allowing cross-training and extending everyone's knowledge of the software system as a whole.

c. How to Improve the Likelihood of Developing Super-conceptualizers

This is a difficult issue and one that is better dealt with from a purely management perspective. It involves the whole notion of power and politics within the Software Support Activity. One of the ways managers achieve power is by controlling information and knowledge like a resource. The Software Support Activity cannot afford to have information tightly controlled. It must be allowed to flow freely. Providing suggested approaches to this issue is outside the scope of this thesis, but it is a critical issue for the Software Support Activity and will require considerable planning and thought.

6. How to Cope With the Different Degrees of Understanding Between Users, Management, and the Maintenance Organization

Communication with users can be improved through visits, direct connectivity (which should include a bulletin board capability), a query formatting assist tool, and the development of troubleshooting and testing procedure guides for the operational sites.

The value of Software Support Activity visits to operational sites and other players is obvious. The connectivity issue is equally so.

It is a key advantage that the Software Support Activity's programmer terminals are networked to allow electronic mail and file transfer. Toshitsugu Nomura [Ref. 39:p. 269] documented the need to not only connect workstation environments within a local area network but also via a wide area network to improve productivity and/or quality improvement. The facilities afforded within the Software Support Activity organization would be equally beneficial if they were distributed to operational sites. Although future Platform connectivity between the Software Support Activity and the operational sites is planned, it is important to ensure that this connectivity includes the capability to computer conference, pass electronic mail, perform file transfer, enable resource sharing, display an electronic bulletin board, and to allow joint document authoring and review.

DEC offers a product called VAXnotes that provides these capabilities. In addition, VAXnotes provides a mechanism to share bug reporting between physically distant locations. An operational site would first check the electronic bulletin board to see if a given

problem had been encountered before and if so access the documented solution. A cross reference tool is built into the VAXnotes that allows the user to browse related topics without having to review the entire bulletin board. If no solution was found, then the operational site could identify the problem and seek advice from other operational sites and the Software Support Activity.

In everyday use, the VAXnotes electronic bulletin board includes a feature that allows a user to view just the notes and updates that the user has not previously seen. The bulletin board has a built-in capability to keep track of what the user has seen in the past.

The electronic bulletin board allows the distribution of expertise. Software Support Activity philosophy and notes to the operational sites can be shared uniformly and quickly, and be readily accessed and reviewed. The electronic bulletin board would also be an effective means to relay and update troubleshooting techniques to the on-site software personnel at each of the operational sites.

Unique features of VAXnotes include a monitoring capability and the ability to restrict some conferences or messages to specific recipients. The monitoring capability is desirable because it allows the named monitor, more than likely the Software Support Activity, to remove certain notes from dissemination that may be offensive or otherwise politically unsound to distribute across the network. Restricted conferencing and message relaying is also desirable. Not all the operational sites will be configured the same or have the same needs. Not all traffic should be shared. The ability to share some

traffic common to all users and restrict distribution of other messages are both needed capabilities.

It is not necessary to buy VAXnotes. What is important is to plan and implement the capabilities of VAXnotes described above. If Platform does not provide file transfer, electronic mail, resource sharing, and a bulletin board capability, then a product like VAXnotes should be considered. A potential DOD alternative is to become a MILNET subscriber of the Defense Data Network (DDN). DDN provides almost worldwide service and a fully capable host always provides electronic mail, file transfer, and resource sharing services. A compartmented traffic capability does exist on DDN if required. DDN may be a viable alternative to consider for the Software Support Activity to meet near term needs.

A continuous problem for the Software Support Activity will be errors and bugs produced from users making incorrect input entries. One way to lessen these errors is for the Software Support Activity to develop a query formatting assist tool. It would be very similar to the templates used in the Language Sensitive Editor. The idea is to provide the user an already pre-formatted shell and help facilities so the user cannot make a mistake. In terms of Brooks' theory, the Software Support Activity would be forcing the user to share the same input view of the software system as the Software Support Activity.

Another way to help the operational sites and the Software Support Activity to share the same model of the software system is for

the Software Support Activity to develop suggested testing procedures and scenarios for the operational sites to use. Vocabulary and how software problems are described will always be a tough issue for both the Software Support Activity and the operational sites. To help identify what the user is trying to describe, the testing procedures can help isolate in what module the error is occurring and the exact behavior the error is exhibiting. The testing procedures would be a valuable communication medium through which Software Support Activity personnel and the operational sites could communicate.

The Software Support Activity must not limit its communication concerns only to inter-organization communication and communication with users, but also must develop aids to help communicate with management. Management is concerned about the bottom line. They are concerned with how long things take and how much things cost.

Boehm, et al. [Ref. 33] suggested that a master project database be defined and implemented to help track these concerns. The master project database would contain "all information relevant to project activities including budget, personnel, scheduling and other managerial data in addition to such technical information as software requirements, design, test procedures, and code...." [Ref 33:p. 34]

In addition, PERT (Program Evaluation and Review Technique) and CPM (Critical Path Method) scheduling and planning tools along with budget analysis tools would be desirable.

A need exists to track problem reporting and change requests from initial recognition to final fix implementation. In addition, there must be some means developed to ensure maintainability aspects and general quality control procedures have been implemented (i.e., documentation updated, code revalidated, new release procedures followed, etc.).

Change is an inherent quality of software maintenance. The Software Support Activity needs a simulation model in order to provide management information on how expensive a particular change may be. Software maintainers make a wide variety of changes, both big and small. It is not always possible to fully determine all the effects a particular change will have on a large program system. The use of a simulation model potentially could allow the results of any change to be visualized. Conceivably, an even more powerful aspect of a simulation model is that it will allow alternatives to be tried and compared.

VIII. CONCLUSIONS

The Software Support Activity tool set has been evaluated and found to be state-of-the-art and of industrial quality. As good as this tool set is as an environment, it does not deal with the problems Brooks' theory predicts are important to software maintenance organizations. In particular, software environments need to help a programmer better understand programs and provide support to the entire software life cycle.

Although VAXset is limited in this regard, software tool environments are beginning to appear on the horizon that may address these problems more adequately. Many in the software industry believe that in the long run what must be developed is a formal language or notation that describes and defines the processes that are taking place. The theme of the 9th Annual International Conference on Software Engineering—Formalizing and Automating the Software Process—further emphasizes this point.

Although the ideas developed in this thesis have been applied to a specific organization, they will apply equally well to any software maintenance organization. In fact, the single most important action the Navy or any other organization may take is to ensure techniques necessary to support software maintenance are included as part of every project's acceptance criteria [Ref. 12:p. 138]. The tools needed in maintenance must be developed during the software development phase. There should be no notion of creating a software maintenance environment after the system has been delivered. Software

maintenance tools must be developed and used in the production environment. There should be no need to make a transition from development to maintenance. Design and maintenance must be coupled more closely. The Navy and other organizations need to incorporate into their philosophy the concept that they should not just ask for an operational system but should ask for an operational system with a software maintenance environment built around it.

APPENDIX A

LIST OF ACRONYMS

CCB	Configuration Control Board
CDD	VAX Common Data Dictionary
CI	Configuration Identification Items
CM	Configuaration Management
CMS	VAX DEC/Code Management System
COMNAVSECGRU	Commander, Naval Security Group
CPM	Critical Path Method
CRLCMP	Computer Resources Life Cycle Management Plan
DDN	Defense Data Network
DEC	Digital Equipment Corporation
DED	Data Element Dictionary
DOD	Department of Defense
ERA	Engineering Research Associates, McLean, VA
FMS	VAX Forms Management System
MSTDF	Mobile System Technical Data Facility
MMS	VAX DEC/Module Management System
NOSC	Naval Ocean Systems Command, San Diego, CA
NTTC	Naval Technical Training Center, Pensacola, FL
PERT	Program Evaluation and Review Technique
SCORE	SIGINT Classification of Recognition of Classified Emitters
SCSS	Shore Cryptologic Support System

SIGINT	Signals Intelligence
SPAWAR	Space and Naval Warfare Systems Command
SQL	Sequel
SSA	Software Support Activity
VAX	Virtual Address Extension
VAXset	VAX Software Engineering Tools
VMS	Virtual Memory System

APPENDIX B

SOFTWARE MAINTENANCE QUESTIONNAIRE

NOTE: "We" means the Software Support Activity.

A. DEVELOPMENT

1. Who is developing the system and under what standards?
   - What were their programming standards? Will we be using the same ones?
   - Requirement and specification standards?
   - Design standards?
   - Source code standards?
   - Documentation standards?

2. Were the developers given specific standards to achieve in order to make the system more maintainable? If not, has the developer tried to improve maintainability by:
   - Setting explicit software quality objectives and priorities
   - Using quality-enhancing techniques and tools
   - Establishing QA activities
   - Choosing a maintainable programming language
   - Improving program documentation
   - Contracting the program

3. Have the developers been late on any phase? Are they on time now?

4. Is anyone on the development team going to be joining the maintenance staff?

5. Are we inheriting any system development which we must complete? How critical is it?

6. Has replacement/retirement and a new release plan been considered?

7. How did the developers ensure their system was easy to understand? That it is:

- Concise
- Consistent
- Complete

8. How were error recovery and restart procedures built-in? (considered minimum required or rich utilities)

9. What was the data flow design method?

10. What was the data structure design method?

11. What other design methods were used?

12. How was the system specified?

- How did NSG sign off on what was developed?
- Specs frozen? When?

B. SYSTEM PARAMETERS

1. System size? How many distinct systems? Number of programs? Number of modules in each? Lines of code? Expected size of data base?

2. Has this system been built in a similar form by the developer?

3. How many output reports does it generate?

4. Reports feed in directly into COMM center or ?

5. What programming language or languages is it programmed in? What major functions are in each language?

- What are they?
- How well integrated are they?
- Did the developers create an integrated environment? If so, is that part of the deliverables?

6. What kind of structure does it have—considered very structured?

7. How much do we have to worry about in terms of efficiency?

8. Which operating system are we using?

9. How real-time critical is the system? How responsive to the analyst must we be?

10. What are the major system components? Are we maintaining all of them? Is there a

- Decision support system?
- Report processing system?
- Information retrieval system?

11. Future evolution expected? In what areas?

12. In what ways is this system distributed?

- How tightly coupled?
- How interdependent?

13. What build-in security restrictions are there?

14. Multithread operations supported?

C. ORGANIZATION

1. How will the personnel be split up?
   - numbers/department roles and functions/expected ability of each? (splitting up according to major functions in the system to be maintained or in major work areas)
   - how will programming teams be organized?
   - how is the user group notion going to be handled?
   - what personnel training will be provided?
2. What is the interplay between workstations vs the VAX?
   - what work will be done on each?
   - how many workstations and VAX systems will there be? How will they be assigned?
3. Where will the civilians be placed? What backgrounds?
4. How often will we be doing in-house re-training?

D. TESTING

1. How has the system done through the various inspections? Who attended?
2. Are change exercises part of the test criteria?
3. Audit checks done when and how?
4. Which programs/modules/systems do the developers consider most error-prone?
5. Programs were completed in what order? Which ones were the most troublesome?
6. Will we be doing any of our own analysis to determine which modules may be the most error prone?
7. Maintenance reviews and testing done how?

8. What was the validation and verification criteria?

9. How is our retesting and revalidation program going to be set-up to validate program changes during maintenance?

10. Are we receiving a complete summary of test results? Does this include information on what specific errors were found and what was changed?

11. What tests were done?

12. Will there be an extended acceptance test where the developers maintain the system for a time while we get to use the system?

13. Did we have maintenance representation at design review?

- Code review?
- Test phase?

14. Everyone tends to make the same kind of errors. Do we know who worked on what module and what their "error style" is?

E. MAINTENANCE

1. What are we really maintaining?

2. How is the term maintenance defined for the organization? In other words, what jobs will the station have?

3. What will be the maintenance philosophy?

4. What are the maintenance objectives?

5. Will a System Maintenance Journal be kept after delivery?

6. Will an Error History be kept?

7. Will a Program Test History be kept and updated (update each time a new version of the program is produced)?

8. What will be the formal change procedure? What is the change control philosophy?

- What will be the process for justifying program change?
- Who will sit on the change review board? How often will it meet?
- Any thoughts to a charge-back system?
- How is quality control ensured during program changes?
- How will maintenance be scheduled?

9. Intend to keep a change-request log?

10. How often do we intend to send updates to our users? Do we have a rough outline what those enhancements will be? If so, what are they?

11. Will we be concerned with configuration management to control hardware, operating system, and utility software changes?

- How do we ensure these items don't get out in the field without us knowing?
- Who is going to keep us informed of changes in this area and what kind of lead time will we get?

12. Are we planning a separate prototype language to make new development/updates of the system or will it be the same? If not, what language has been chosen?

F. DOCUMENTATION

1. What documentation did we ask for? What form is it in? Specifically what deliverables in each of the four classes of documentation:

- user documentation
- operations documentation

- program documentation
- data documentation (i.e., data model and data dictionary)

2. Do we have on-line user documentation?
   - on-line help facilities—ability to inquire about each user function
   - computer-aided instruction

3. Besides internal documentation, what external documentation will be delivered?
   - When was the last time it was updated?
   - Does it represent the delivered system or earlier versions?
   - What check was used to ensure this?
   - Will we be provided a history of changes?

4. Is a system development journal being turned over?
   - What were the original design intentions?
   - What parts of the system did the developers consider the most difficult?
   - What was the development philosophy?
   - Will we be provided reasons why the developer selected particular designs?
   - Will we be given information concerning what designs were contemplated and reasons why they were rejected?
   - What were the project goals and priorities?
   - What experimental techniques and tools were used?
   - What day-to-day problems did they encounter?
   - What do the developers consider their project successes and failures?
   - What went right? What went wrong?

5. Is an Error History being turned over?

6. A Program Test History?

7. To what degree are we going to have to re-document the system we've been delivered? Any thought to re-documenting the system as a learning exercise for the station prior to taking over control?

8. What documentation standards did the development organization use?

9. How were the following things documented?

- source code
- overview
- program organization
- control structure
- program comments
- instruction level comments
- meaningful names
- code style

G. TOOLS

1. What environment are we using?

2. Are tools used in development part of the deliverables?

3. How would I find out what tools the developers are using?

4. What hardware and software did the developers use in developing the system?

5. Test data and test drivers—are we getting them? Do we know what the developers used?

6. What data administration techniques will be used? (i.e., control design and definitions of all data?)

7. What logging and audit tools are part of system? (i.e, automatic audit trails, accuracy controls, logs of usage)

8. Will there be a procedure library?

9. Was the system built with defensive programming aids built in, then removed by optimization techniques? If it was, are we getting the tools to input defensive code and to optimize the code for operational usage? If not, will we be building these tools?

10. Would a potential goal of the organization be to develop an integrated environment?

H. USER

1. How often do we intend to make visits out to the sites?
2. Philosophy on user enhancements:
   - Separate department to handle enhancements?
   - Batch?
   - Cost-back scheme?
   - Who decides what changes?
   - Who will be on the Change Control Board?
3. What user training will we be doing?
4. What hardware will the user have?
   - Others possible?
   - Are they connected to other sites? How much information will they be passing? How consistent must the data bases be between sites?
5. Will end users be maintaining user documentation and their own user training?
6. How are we going to improve the user's understanding of how to more effectively use the software system?
7. Will the user be making software changes?

8. To what degree are the users going to be the ones responsible for making updates?

I. BUDGET

1. What does the budget look like in terms of:
   - further training
   - travel to user
   - money for software tools
   - software and hardware enhancements

J. REPORTING

1. What information will we provide users?
2. What will users be reporting to us and how often?
3. Who do we report to and how often?
4. How many output reports? How easy will their format be to change?
5. Will users have a good report generator tool available to adapt their reports or will we disallow this?
   - Are there mapping needs?
   - Graphics needs?

K. DATABASE ISSUES

1. Are we maintaining data management facilities?
2. Who will be the data administrator? How will he monitor data models used within each segment?
3. How has the need for data independence been assured?
4. I understand we are using ORACLE. What relationship will be maintained between the data base, ORACLE, and VAX/VMS?

5. What constraints have been imposed on the system to prevent disintegration of the database from class 3 to class 2?

6. Are we using application development without programmers? (What I mean is, will the sites be allowed to develop some SQL queries on their own or must all the query manipulation programs be blessed by us first?)

7. Is the system a combination of a relational data base with an information retrieval system? (joined or separate)

8. What structure or structures are used to access the data? (i.e., search and join, secondary indices, ring structures, or ?)

9. What constraints are built-in to limit/avoid redundancy?

10. How stable is the data?

11. Is this a single data base or multiple data base system?

12. Data stability:
    - Have we identified the data model?
    - Able to add files?
    - Able to create new access paths?
    - Do we have automatic generation of data descriptions from a dictionary? Any capability to prevent programmers from inventing their own data descriptions?
    - Ability to change associations among records?
    - Flexible query facilities and report generators?
    - Are we allowing application generators at local sites (ability to generate applications from the data base without programmers)?
    - Get results via command (like query) vice writing a program?
    - Data at each site will be different or not? Was the consideration built-in?

13. What kind of usage are we expecting? A couple of terminals off the system? What?

14. To what degree did we do data modeling?

- Developers only?
- Did we send Navy personnel to help?
- Has a canonical model (computerized tool that helps in building data models) of data been created?
- Do we have a data model standard to be used by all sites?
- Is there a standard naming convention for selecting data-item names?

15. Are data dictionary and data modeling tools included in the deliverables?

- Is the data dictionary built into the DBMS?

16. Have we identified future data needs?

- Can we change key fields or are keys used at all?

17. What kind of data base is it?

- subject data base
- isolated application data bases
- information system data bases

18. How complex is the query language?

19. Why did we elect to go with a tailored system as opposed to using a commercial system?

20. How adaptable is the system to "What if" questions? How flexible a system is it to new associations? To what degree are we going to allow the user to use their imagination and tailor the system to their needs?

21. How independent is the way data is stored to the way it is used?

22. Interoperability with NSA data bases? Other service data bases? Wizard?

L. FUTURE

1. How will we be set-up so as to determine long-term future growth and potential system replacement?

2. How will we do strategic planning?

APPENDIX C

SOFTWARE SUPPORT ACTIVITY TOOL SET

A. VMS SOFTWARE DEVELOPMENT ENVIRONMENT

The VMS Software Development Environment has been bought for the Software Support Activity. It is an integrated package that was developed specifically to increase programmer productivity, increase product quality, help manage complexity, and increase the effectiveness with which programmers implement, test, and maintain programs.

The VMS Software Development Environment can be broken into four basic categories: the VMS operating system, VAX languages, VAXset, and related VAX software.

The VMS operating system is a general-purpose operating system [Ref. 40:p. 1-5]. An operating system is responsible for the coordination and management of a computer system's resources. The VMS operating system is the foundation upon which the rest of the software development environment rests. It serves as the focus point and the driver of all VMS software components. Since all resources are compatible with each other and have been designed to work together, the specific services and utilities of the VMS operating system may be invoked directly by VAXset tools. [Ref. 36:p. 1-1]

The sixteen VAX languages include Ada, APL, BASIC, BLISS, C, COBOL, DIBOL, FORTRAN, Pascal, PL/I, and RPG II. The VAX version of FORTRAN and Pascal will be used by the Software Support Activity.

An important capability to note concerning the sixteen languages is that each is capable of calling programs written in another VAX language. [Ref. 40:p. XVI]

VAXset is comprised of five software tools: VAX Language-Sensitive Editor, VAX Performance and Coverage Analyzer, VAX DEC/Test Manager, VAX DEC/CMS, and VAX DEC/MMS. Each will be covered in more detail later in this appendix.

Related VAX software includes capabilities for data communication, information management, and cross development. Each of these capabilities is optional. Of them, the Software Support Activity will only make use of the data communication (DECnet) facility.

The VAX/VMS Software Development Environment was designed to be an integrated environment. All components of the environment already described were designed to a common specification and were based on a single operating system (VMS) and on the same architecture (VAX).

The common specification is termed the VAX Common Language Environment. It standardizes calling conventions, condition and error handling, and programming practice.

It is the VAX Common Language Environment that allows programs written in one VAX language to call other programs written in a different VAX language. The VAX Common Language Environment also allows the VAXset tools to communicate with each other by means of a compatible set of data formats. In addition, the VAXset tools share a common user interface. The tools are consistent in terms of user

input and response to that input. They share the same command language, prompts, and error messages. An important characteristic of VAXset tools is many of the tools may be customized and extended to meet user requirements. [Ref. 40:pp. 1-1 to 1-2]

B. VAXSET

1. Language-Sensitive Editor

The Language-Sensitive Editor allows you to write programs in the VAX language of your choice. It is a multi-window screen-editor that is non-modal. It allows the completion of many programming tasks within a single editing session. Programmers can write, edit, compile, review diagnostics, and correct compilation errors without ever leaving the editor. [Ref. 36:p. 1-24]

The editor has a built-in understanding of the syntax of the programming language being used. It provides pre-formatted templates to help program development and offers on-line help facilities. [Ref. 40:p. 1-17]

The templates are formatted language constructs that contain all the key syntactic elements. User input areas are indicated by required or optional placeholders. The user may input program text directly into a placeholder or choose a given option from a provided menu.

Users can tailor the templates to match the programming standards of the organization. Templates also can be created for documentation standards, since the Language-Sensitive Editor is a text-oriented editor.

The help facilities provide extensive language-specific information and specific help in using the language-specific templates.

The editor is directly compatible with many of the VAX languages (can invoke the VAX compilers directly), the VAX Debugger, and the VAX Performance and Coverage Analyzer. Since it is compatible, the Language-Sensitive Editor may be directly invoked from the Debugger or the Performance and Coverage Analyzer. The VAX Source Code Analyzer and the VAX DEC/CMS, on the other hand, can be directly invoked from the editor. [Ref. 40:pp. 1-17]

2. VAX Performance and Coverage Analyzer

The VAX Performance and Coverage Analyzer can be used to fine-tune and optimize source code for peak efficiency. It is suitable for finding performance hot spots and ensuring thorough test coverage.

The VAX Performance and Coverage Analyzer consists of two parts—the collector and the analyzer. The collector gathers all performance or test coverage data from an executing program. The analyzer uses the information collected by the collector to produce histograms and tables showing the parts of a program that consume the most resources. The type of information that can be displayed is an indication of what part of a program takes the most time, page fault data, what VMS services are called and how often, I/O usage data, and what paths are exercised as part of your test coverage. The information displayed can be produced at a very detailed level or at a very coarse level. [Ref. 36:pp. 1-24]

3. VAX DEC/Test Manager

The VAX DEC/Test Manager provides automatic and consistent software testing. It helps the user manage the testing process by organizing collections of user-designed tests. The DEC/Test Manager allows a user to select tests, run them, and verify and review results. Tests can be created either interactively, or via DCL (standard VAX/VMS command language interface) command scripts. [Ref. 40:pp. 1-16 to 1-17]

The DEC/Test Manager is an automated regression testing system that can be used throughout the software life cycle. It automatically executes user-defined tests and compares test output against pre-established benchmarks (VAX DEC/CMS can be used for the storage of test templates and results). The benchmarks are either supplied directly by the user or are benchmarks stored from a previous test run. [Ref. 40:pp. 1-16 to 1-17]

The DEC/Test Manager can continue to be used even after existing features have been updated or new features added. The Test Manager includes a feature that can predict expected results. If the expected and actual results differ greatly, then the software has regressed and needs to be fine tuned. [Ref. 40:pp. 1-16 to 1-17]

The VAX Performance and Coverage Analyzer and the VAX DEC/Test Manager can be used together to run a complete group of tests on an entire software system. The use of the VAX Performance and Coverage Analyzer in this case is automated under the VAX

DEC/Test Manager. This capability can be used to ensure any new code written has been adequately tested. [Ref. 40:pp. 1-16 to 1-17]

4. VAX DEC/Code Management System (CMS)

The VAX DEC/CMS is a program librarian. It is used to track all changes made to a program's source code file including ancillary information of who made the change, why they made the change, and when. [Ref. 36:p. 1-22]

The VAX DEC/CMS stores both current and historical versions. Therefore, it can be used to both reconstruct prior versions and to identify and freeze software for release. [Ref. 36:p. 1-22]

Its most powerful feature is the ability to either prohibit or permit concurrent reservations. In other words, CMS can either be configured not to allow two programmers to work on the same segment of source code or it can be configured to allow two or more programmers to make changes. If concurrent reservations are prohibited, then it allows users to reserve files for their exclusive access. On the other hand, if concurrent reservations are permitted, CMS has the capability to keep track of all edits performed by two or more project members working on the same files at the same time. When this occurs, CMS notifies each programmer that someone else is working on the same code segment. If the changes do not conflict, then CMS can without intervention merge all changes. If, when the merge is completed, code conflicts have occurred, then CMS notifies the users of the conflict and identifies the problem in a difference file. The programmers may or may not take action depending on the use of

their code segment. (Note: CMS can allow multiple versions of a source file provided it is linked to a specific version of the target software system.) Thus, users do not have to worry about undoing someone else's work or making changes that may adversely affect someone else's files.

As mentioned previously, CMS can be used directly from the Language Sensitive Editor.

5. VAX DEC/Module Management System (MMS)

The VAX DEC/MMS is used to manage system builds. It makes easy the maintenance of current versions of routines, modules, or files that have undergone many changes. MMS ensures the current version includes all the latest changes and interdependencies. It also rebuilds systems efficiently since it only updates those components that have changed since the last build. If MMS does not have a given routine, module, or file, it is smart enough to obtain access to the files needed directly from VAX DEC/CMS libraries or from VAX/VMS libraries. [Ref. 40:p. 1-16]

APPENDIX D

SOURCE CODE ANALYZER

The Source Code Analyzer provides support for seven languages including FORTRAN and Pascal. It provides three basic capabilities: source code cross-referencing, code navigation or browsing, and static analysis.

The Source Code Analyzer can be invoked directly from the Language-Sensitive Editor. Both tools are tightly integrated through an analysis file with the VAX compilers. The analysis file is used to create a Source Code Analysis Library that is essentially a cross-reference database. At the end of every compilation, any new or changed cross-references are merged with all previous cross-references created during earlier compiles. The Source Code Analysis library uses hashing and indexing to allow fast access to the cross- referenced data.

Typically, the Source Code Analyzer will be used from the context of the Language-Sensitive Editor. The user defines a symbol to be cross-referenced within the source code. The upper portion of the screen contains a table of information about the symbol selected. The table is divided into four parts: Symbol Name, Class (data type, i.e., variable, constant), Module/Line (location of symbol occurrence), and Type of Occurrence (read reference, write reference, etc.). Specification of the symbol to be cross-referenced may be either exact or vary in the degree of specification through the use of a wildcard (*) symbol.

All symbols in the cross-reference database may also be selected, but it naturally is extremely slow to list all symbols. An editing window (the Language-Sensitive Editor) and a command line make up the bottom portion of the screen. The table information and the source code, in the editing window, are visible at the same time.

Navigation through the source code is possible by selecting the location information or the type of occurrence of a given symbol. Depending on what was chosen, the source code within the editing window is updated to reflect the location or occurrence desired. Declaration information related to the symbol can be requested and pops up within its own window just above the source code. The declaration of a symbol and its appearance in source code can thus be readily compared.

Navigation is not only possible from the cross-reference table to source code but also from source code to cross-reference information. If a symbol is selected within the source code that is not currently reflected in the cross-reference information, then the table will be updated. Moving from source code to cross-reference information and back again is very easy and natural for the user.

The breakout of type of occurrence information into one of the following: read reference, write reference, address reference, variable declaration, constant declaration, and formal parameter declaration, is extremely powerful for a maintenance programmer who does not know much about the code he is currently viewing and in which he needs to browse around. Depending on the program error, write

references (when a variable is changed) may prove the most likely source of the error. What is learned during the review of the write reference occurrences would dictate where the next most likely occurrence of the error may be.

The static analysis portion of the Source Code Analyzer consists of two portions: the ability to view the call tree and a means to check calling argument consistency.

The view of the call tree may be either specified or limited to a particular depth. The Source Code Analyzer will look through the cross-reference database and find all calls that came from the referenced routine, module, function, or procedure. This part of the static analysis tool will also indicate whether the calls referenced are recursive calls or not.

The check calls command determines whether any calls are not consistent. For example, the check calls command checks that all the procedure calls' type declarations and number of parameters match.

LIST OF REFERENCES

1. Charette, R. N., Software Engineering Environments: Concepts and Technology, Intertext Publications, Inc., 1986.

2. Reifer, D. J. and Trattner, S., "A Glossary of Software Tools and Techniques," Computer, v. 10, pp. 52-60, July 1977.

3. Howden, W. E., "Contemporary Software Development Environments," Communications of the ACM, v. 25, pp. 104-135, May 1982.

4. Rome Air Development Center Technical Report RADC-TR-85-112, A Taxonomy of Tool Features for a Life Cycle Software Engineering Environment, by E. S. Kean and F. S. LaMonica, June 1985.

5. National Bureau of Standards Publication NBS-IR-80-2159, NBS Software Tools Database, by R. Houghton and K. Oakley, 1980.

6. Fjeldstand, R. K. and Hamlen, W. T., "Application Program Maintenance Study: Report to Our Respondents," as taken from Tutorial on Software Maintenance, by G. Parikh and N. Zvegintzov, IEEE Computer Society, pp. 13-27, 1983.

7. EIA Configuration Management Bulletin No. 4A, Configuration Management for Digital Computer Programs, by G-33 Configuration Control Committee, p. 2.22, April 1979.

8. Naval Electronic Systems Command UNCLASSIFIED Letter 9470: PDE 107 Serial 9C6/2889 to Distribution, Subject: Promulgation of the Shore Cryptologic Support System (SCSS) Computer Resources Life Cycle Management Plan (CRLCMP) Volume I, 23 August 1984.

9. Martin, J. and McClure, C., Software Maintenance: The Problem and Its Solutions, Prentice-Hall, Inc., 1983.

10. Buxton, J. N. and Druffel, L. E., "Rationale for Stoneman," as taken from Interactive Programming Environments, by D. R. Barstow, H. E. Shrobe, and E. Sandewall, McGraw-Hill Book Co., pp. 535-545, 1984.

11. Mitchell, C. Z., "Engineering VAX Ada for a Multi-Language Programming Environment," Proceedings of the ACM SIGSOFT/ SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Association for Computing Machinery, pp. 1-10, December 1986.

12. McClure, C. L., Managing Software Development and Maintenance, Van Nostrand Reinhold Co., 1981.

13. Swanson, E., "The Dimensions of Maintenance," Proceedings of the 2nd International Conference on Software Engineering, IEEE Computer Society Press, pp. 492-497, October 1976.

14. Parikh, G. and Zvegintzov, N., Tutorial on Software Maintenance, IEEE Computer Society, 1982.

15. Fairley, R. E., Software Engineering Concepts, McGraw-Hill Co., 1985.

16. Heninger, K. L., "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," IEEE Transactions on Software Engineering, v. SE-6, pp. 2-13, January 1980.

17. Schneiderman, B., Software Psychology, Winthrop Publishers, 1980.

18. Lientz, B. P. and Swanson, E. B., Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations, Addison-Wesley Publishing Co., 1980.

19. Shneiderman, B. and Mayer R., "Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results," International Journal of Computer and Information Sciences, v.8, pp. 219-238, March 1979.

20. MacLennan, B. J., Principles of Programming Languages: Design, Evaluation, and Implementation, Holt, Rinehart, and Winston, 1983.

21. MacLennan, B. J., Functional Programming Methodology, to be published by Addison-Wesley, 1987.

22. Miller, G. A., "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," The Psychological Review, v. 63, pp. 81-97, March 1956.

23. Brooks, R., "Towards a Theory of the Comprehension of Computer Programs," International Journal Man-Machine Studies, v. 18, pp. 543-554, 1983.

24. Brooks, R., "Using A Behavioral Theory of Program Comprehension in Software Engineering," Proceedings of the 3rd International Conference on Software Engineering, IEEE Computer Society Press, pp. 196-201, 1978.

25. Brooks, R., "A Theoretical Analysis of the Role of Documentation in the Comprehension of Computer Programs," Proceedings of Human Factors in Computer Systems, pp. 125-129, 1982.

26. Wiedenbeck, S., "Processes in Computer Program Comprehension," as taken from Empirical Studies of Programmers, by E. Soloway and S. Iyengar, Ablex Publishing Co., pp. 48-53, 1986.

27. Curtis, B., Tutorial: Human Factors in Software Development, IEEE Computer Society, 1981.

28. Weiser, M., "Program Slicing," Proceedings of the 5th International Conference on Software Engineering, IEEE Computer Society, pp. 439-449, 1981.

29. Weiser, M., "Programmers Use Slices When Debugging," Communications of the ACM, v. 25, pp. 446-452, July 1982.

30. Boehm, B. W., "Seven Basic Principles of Software Engineering," Journal of Systems and Software, v. 3, pp. 3-24, 1983.

31. MacLennan, B. J., Programming Tools and Environments, draft, 1987.

32. Curtis, B., and others, "On Building Software Process Models Under the Lamppost," Proceedings of the 9th International Conference on Software Engineering, IEEE Computer Society, pp. 96-103, 1987.

33. Boehm, B. W., and others, "A Software Development Environment for Improving Productivity," Computer, v. 17, pp. 30-42, June 1984.

34. Davis, W. S., Systems Analysis and Design: A Structured Approach, Addison-Wesley Publishing Co., 1983.

35. VAX/VMS Software Information Management Handbook, Digital Equipment Corporation, 1985.

36. VAX/VMS Software VMS System Software Handbook, Digital Equipment Corporation, 1985.

37. Balzer, R., "A 15 Year Perspective on Automatic Programming," IEEE Transactions on Software Engineering, v. SE-11, pp. 1257-1268, November 1985.

38. Yourdan, E., Managing the Structured Techniques: Strategies for Software Development in the 1990's, Yourdan Press, 1986.

39. Nomura, T., "Use of Software Engineering Tools in Japan," Proceedings of the 9th International Conference on Software Engineering, IEEE Computer Society Press, pp. 263-269, 1987.

40. VAX/VMS Software Language and Tools Handbook, Digital Equipment Corporation, 1985.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Professor G. Bradley, Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, California 93943	5
4.	Professor B. J. MacLennan, Code 52Ml Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
5.	LT J. Sexton 16 Madison Ave. New Providence, New Jersey 07974	5
6.	Chief of Naval Operations Director, Information Systems Navy Department (OP-945) Washington, D.C. 20350-2000	1
7.	Officer in Charge Software Support Activity Corry Station Naval Security Group Detachment Pensacola, Florida 32511-5000	2
8.	Commander Naval Security Group Command Attn: G30 Naval Security Group Headquarters 3801 Nebraska Ave., N. W. Washington, D.C. 20390-5210	1

Thesis
S41935 Sexton
c.1 Software tool selection
for U.S. Navy software
maintenance organization.